FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Automated Measurement Method for the Usability of APIs

and its Application in the Area of Middleware

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor/in der technischen Wissenschaften

by

Thomas Scheller
Registration Number 0225689

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. eva Kühn

The dissertation has been reviewed by:

(Ao.Univ.Prof. Dipl.-Ing. Dr. eva Kühn)

(Univ.Prof. Mag. Dr. Manfred Tscheligi)

Wien, 26.08.2014

(Thomas Scheller)

Erklärung zur Verfassung der Arbeit

Thomas Scheller
Lagergasse 31/10, 3425 Langenlebarn

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

This work would not have been possible without the support of other people, who should not be left unmentioned.

First I want to thank my supervisor eva Kühn for her continuous support and for always having time for me and my work, despite of her always overloaded time schedule.

Second I want to thank Robert Pabeschitz for always putting his trust in me, and for enabling a combination of work and research that helped me a lot on my way.

Special thanks also go to all the participants of my usability studies, as well as to the Interactive Media Systems Group at TU Vienna for lending us their eye tracker – without them this work would not have been possible.

Last but not least I want to thank my wife Bernadette for always supporting me, and for being a source of strength and inspiration.

Abstract

Usability is an important quality factor for every kind of software product. Usable software increases productivity and reduces cost and errors, while software that is difficult to use causes worry and frustration, and discourages further use. Because of that, usability tests have become an integral part in the development of software products that provide any kind of user interface. For the special area of application programming interfaces (APIs), this is unfortunately not yet the case. APIs are used by developers to access the functionality of external software libraries, which they want to utilize when writing their own software. A large and rapidly growing number of such APIs is available for a variety of tasks, e.g. for logging, database access or unit testing. So, many developers are using APIs on a daily basis, which makes their usability an important attribute.

While many methods have already been introduced for measuring usability, like Heuristic Evaluation and Thinking Aloud, most of these methods are themselves not easy to use: They require experienced evaluators, and the results also strongly depend on their opinions. Many methods require an already fully functional product, and a usability lab with sufficient equipment is needed for running user tests. This makes tests cost- and time-expensive.

The goal of this thesis is to treat these problems, by creating an automated and objective API usability measurement method. It is based on evaluating the API structure as well as API usage examples. By providing an API developer with fast initial response to the designs of his/her APIs, it can help to avoid costly changes in late development stages. Further, by calculating comparable usability values it allows to objectively compare the usability of different APIs. Because no experienced evaluators are needed, the method can also be used by developers that are inexperienced in this area, or cannot afford costly usability tests. All this makes the method a useful addition to existing usability measurement methods.

In this thesis, we conduct a comprehensive literature review of existing usability evaluation methods as well as automated software metrics, to show that such a method does not yet exist. We then conduct two usability studies to find factors influencing usability that can be measured automatically. We also review existing studies, guidelines and reports about API usability problems to find potential usability factors. From the resulting data, we extract measurable concepts and properties, using statistical evaluation methods. We then construct an automated usability measurement method based on these properties, called the “API Concepts Framework”. Finally, we apply the framework to APIs in the area of communication middleware, and present a middleware that is optimized for usability. We show that the results provided by the framework are valid and provide useful information for rating, comparing and improving and API’s usability.

Kurzfassung

Usability ist ein wichtiger Qualitätsaspekt für jede Art von Software. Software, die einfach zu verwenden ist, erhöht die Produktivität, verringert Kosten und vermeidet Fehler. Schwierig zu verwendende Software führt zu Frustration und Ablehnung. Um User Interfaces auf Ihre Benutzbarkeit hin zu überprüfen, werden Usability Tests daher immer mehr zum integralen Bestandteil eines jeden Softwareentwicklungsprozesses. Für eine spezielle Art von User Interface ist dies jedoch leider noch kaum der Fall: Application Programming Interfaces (APIs). APIs werden von SoftwareentwicklerInnen verwendet, um auf die Funktionalität von externen Softwarebibliotheken zuzugreifen, welche sie verwenden, um Ihre eigene Software zu bauen. Eine immer größer werdende Anzahl solcher APIs ist für verschiedenste Aufgaben verfügbar, wie z.B. Logging, Datenbankzugriff oder Unit Testing. Viele EntwicklerInnen verwenden täglich solche APIs, was deren Usability zu einem wichtigen Aspekt macht.

Während bereits viele verschiedene Methoden zur Messung von Usability existieren, wie zum Beispiel Heuristic Evaluation und Thinking Aloud, sind die meisten dieser Methoden leider nicht einfach zu verwenden: Sie erfordern Usability-Experten für die Evaluierung, und oft auch ein bereits voll funktionsfähiges Produkt, sowie ein gut ausgestattetes Usability Test Labor. All dies macht Tests sowohl Kosten-, als auch Zeit-aufwändig.

Ziel der vorliegenden Arbeit die Behandlung dieser Probleme durch die Definition einer automatisierten und objektiven Usability Messmethode für APIs. Sie erlaubt es, dem/der EntwicklerIn schnell und früh Rückmeldung über die Usability ihrer/seiner APIs zu geben, und hilft dadurch, teure Anpassungen in späteren Entwicklungsphasen zu vermeiden. Durch die Berechnung vergleichbarer Usability-Werte ist es weiters möglich, verschiedene APIs objektiv zu vergleichen. Da keine Experten für die Auswertung notwendig sind, kann die Methode auch von Entwicklern angewendet werden, die keine Erfahrung in diesem Bereich besitzen, oder sich keine aufwändigen Tests leisten können.

In der vorliegenden Arbeit zeigen wir mit einer ausführlichen Literaturrecherche, dass eine solche Methode noch nicht existiert. Darauffolgend führen wir zwei Usability Studien durch, um Faktoren zu finden, die die Usability von APIs beeinflussen. Wir untersuchen außerdem existierende Studien, Richtlinien und Berichte über API Usability. Aus den Ergebnissen extrahieren wir messbare Konzepte und Eigenschaften, welche wir mittels statistischer Auswertungsmethoden evaluieren. Basierend darauf erstellen wir eine automatisierte Usability Messmethode. Abschließend verwenden wir die neu entwickelte Methode, um APIs im Bereich Kommunikations-Middleware zu bewerten, und präsentieren eine Middleware mit optimierter Usability. Wir zeigen, dass die Methode nützlich ist, um die Usability von APIs zu bewerten, zu vergleichen und zu verbessern.

Contents

CHAPTER 1

Introduction

Usability is an important quality factor for every kind of software product. The ISO 9126 standard [103] defines it as "the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions". Software that is difficult to use wastes the user's time, causes worry and frustration, and discourages further use. It also increases the cost of training and support. On the other hand, usable software increases productivity and reduces costs. It satisfies and motivates users.

Nowadays software developers are facing the situation that more and more people have access to their applications. For example, popular apps on app stores for mobile devices are accessed by millions of people. For software developers this means that a large variety of users may potentially use their software, and they need to make sure that it is easy to use for all potential users. Because of that, usability tests are an integral part of the development of a software product that provides any kind of user interface.

Unfortunately, in a special area of user interfaces, conducting usability tests is not yet a standard procedure: the area of application programming interfaces (APIs). Software developers use APIs to access the functionality of external software libraries, so that they can utilize it in their own software. That is an important difference to typical user interfaces: The user of the API is not the end user of the software product, but the developer that uses the API to create a software product. Yet that doesn't change the importance of usability, as [148] says fittingly: "Imagine, hypothetically, just for a moment, that programmers are humans."

APIs of external software libraries are used for a variety of tasks, like: Logging (e.g. log4j[1]), database access (e.g. Hibernate[2]), unit testing (e.g. jUnit[3]) and remote communication (e.g. NServiceBus[4]). A recent analysis of open-source Java projects [121] reports that about 4-6 different external libraries are used per project, with some projects even using more than 20. In addition to external libraries, a software product itself is often developed as multiple components,

[1]http://logging.apache.org/log4j/
[2]http://www.hibernate.org/
[3]http://junit.org
[4]http://www.nservicebus.com/

which is known as component based software engineering, or CBSE [25]. These components are communicating with each other via their public APIs. Especially in larger software development teams where different developers are working with the components, the component APIs need to be defined with usability in mind.

Particularly with APIs for remote communication (also called *middleware*), we discovered severe usability problems. Many middleware solutions offer the user a lot of functions and configuration options to deal with the complex problems of remote communication, and because of that are very hard to learn and understand. Others try to hide the complexity from the user by adopting local communication paradigms like method calls (e.g. Java RMI [49]), but by hiding important aspects of remote communication like asynchrony from the user they make it difficult to implement more complex scenarios. Further, middlewares following alternative paradigms like Space Based Computing [43, 74] provide elegant solutions for various communication scenarios, but are hard to understand because they are completely different from anything the users know.

Several reports confirm the importance of usability for APIs [5,37,94,133,157,163]. Things like bad/missing documentation and usage examples, unintuitive structure and names, or functions with unexpected behaviour, can have significant impact on usability. Such problems can lead to the developer using functions in a non-optimal or completely wrong way, or even in being unable to complete a task with the API at all. Unfortunately, evaluating the usability of APIs is not an easy task.

1.1 The Problem of Measuring API Usability

Many methods have already been introduced for measuring usability in general. Some of the most prominent are Cognitive Walkthroughs [152], Heuristic Evaluation [100], Thinking Aloud [142] and Surveys [142]. Although most of these methods can generally be applied to every kind of "software product", their applicability for the special area of APIs has not been sufficiently researched. E.g. for heuristic evaluation, it is likely that other heuristics will be needed for an API than for a graphical user interface (GUI).

Further, most of these methods are themselves not easy to use: They require experienced evaluators, and the results also strongly depend on their opinions. This can lead to wide differences depending on the evaluator, and is called the "evaluator effect". [96] shows that this effect persists across different evaluation methods, problem domains, system complexity, prototype fidelity and problem severity, and can be as strong as two different evaluators agreeing in only 5% of the found usability problems. This also means that a majority of the problems can only be found with multiple evaluators.

Many methods require a fully functional product, which is not yet available in earlier design stages where feedback on the user interface design would be needed most. For many tests also a representative set of users is needed. For APIs there is less user diversity than for other kinds of user interfaces, because users are typically software developers. This makes the influence of factors like age and education negligible. But a factor that influences usability can be how a developer approaches an API. [37] names *opportunistic*, *pragmatic* and *systematic* as three different kinds of approaches. Another factor is the target domain of the API. Developers working in different domains can have very different knowledge and expectations on APIs they need to use. Therefore test users need to be well chosen with various factors in mind.

To execute user tests, a test lab is needed with equipment for running the tests and recording results. To distract the user as little as possible, but still acquire a maximum amount of data, test labs often have expensive installations, like video cameras, eye tracking equipment and a control room with a one way mirror. All this makes tests very cost- and time-expensive, and difficult to integrate into a software engineering process.

These problems apply especially to non-commercial projects, where neither financial resources nor enough time is available for hiring experienced evaluators and conducting extensive usability studies. A good example for such a project is Json.Net[5], a library for serializing .Net objects in the JSON format [44], which is developed by a single person. When looking into the NuGet Gallery[6], which is a popular repository for third party .Net libraries, Json.Net has been downloaded more than 1.5 million times, making it the second most popular library available on NuGet. In such a case, usability studies could bring improvements for a large number of users, but existing methods are nearly impossible to use in this kind of project, because of the above mentioned reasons.

Evaluating the usability of an API would not only be useful for API developers, but also for API users, to get a decision factor when planning to use one of multiple external APIs. For example, when different middlewares are available that could potentially be used for a certain use case, and all of them fulfill the necessary functional requirements, usability could be a central decision factor. Next to requiring a lot of effort, a problem with existing measurement methods is that the results are mainly suggestions for where improvements could be made, but are hard to quantify, and are therefore not suitable for comparison. To produce comparable results, an even larger number of test users would be needed [86], as well as enough evaluators to minimize the evaluator effect. Further, comparable tests would need to be carried out for all considered APIs. Since this much effort could never be justified just for choosing an appropriate API, usability is rarely used as a decision factor.

1.2 The Unsuitability of Code Complexity Measures

There is a number of existing measures (or *metrics*) that can be applied to measure different aspects of code complexity. Popular examples are the cyclomatic complexity metric [132] or the object-oriented software metrics by Chidamber & Kemerer [32]. Such measures are often used by tools (e.g. NDepend [179], a popular software measurement tool for .Net) to help developers evaluate their own code. Since APIs are written and accessed with code in a certain programming language, their usability is of course also related to the complexity of code. Because code complexity measures allow an automated execution, they could solve some of the problems of usability measurement methods, most of all the high costs.

Unfortunately, such measures are not suitable for measuring usability: Most of the measures, like the two mentioned above, deal with the inner structure of code, but not with the publicly exposed API. For example, for users of the API it is irrelevant how complex the implementation

[5]http://james.newtonking.com/projects/json-net.aspx

[6]http://nuget.org/

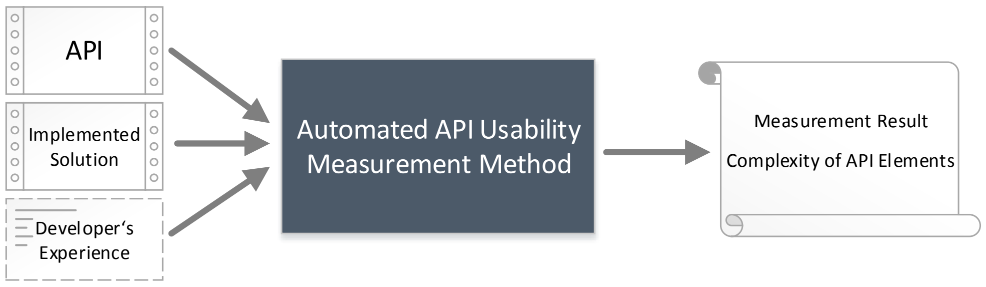

API
Implemented Solution
Developer's Experience
Automated API Usability Measurement Method
Measurement Result
Complexity of API Elements

Figure 1.1: Inputs and outputs of an automated API usability measurement method

of the API's methods is, or how many non-public classes there are. Only two recent publications [13,153] deal with the specific topic of measures for API usability. But they both suffer from two significant problems: First, they mainly measure properties of an API's methods (e.g. number of method parameters), which will not be enough for APIs that rely on other programming concepts, like annotations. Second, they do not take the context of use into account, which is described as a central factor in all popular usability definitions (like [103]).

1.3 The Solution: An Automated Usability Measurement Method

As a solution to these problems, in this thesis we propose an automated and objective usability measurement method for APIs, that takes the context of use into account. This can be done by not only analyzing the API itself, but also the code of the implemented solution and the experience of the developer that used the API (which represent the context of use). The inputs and outputs for such a method are illustrated in Figure 1.1: As input, the measurement method takes the API to be analyzed (e.g. a jar file in Java, or an Assembly file in .Net), the code of a solution that has been implemented using this API, as well as (optionally) information about a developer's experience in a predefined format (e.g. how often he/she has used the API already). As output the method provides an overall complexity value, as well as information about the complexity of the used API elements (classes, methods, etc.). The detailed results can help API developers finding out which aspects of the API need improvement. The overall result can be used for comparison with other APIs, when a solution is implemented for the same problem. Since the measure is automated, no evaluators are needed, preventing the "evaluator effect" and making the results comparable. Further, for an evaluation of the API it is not necessary that its functionality is implemented – it can be evaluated as soon as the API design is finished and usage examples have been created. So, API developers can get early feedback concerning the usability of their API design.

To summarize, the reasons why an automated and objective usability measurement method for APIs would be important, and a useful addition to existing measurement methods, are:

- For API developers, to get fast initial response to the designs of their APIs, and to avoid costly changes in late development stages

- For API users, to be able to objectively compare the usability of different APIs, which is not possible with existing methods, since they don't provide quantifiable results (except for A/B tests [87], but they require a lot of effort)
- For both API developers and users, to make usability evaluation possible even if they are inexperienced in this area, or cannot afford costly usability tests

Existing usability measures cannot fulfill theses points because of the mentioned problems. Of course, just like software complexity measures cannot completely replace personal code reviews, such kind of automated usability measure can never completely replace a thorough usability investigation with human evaluators or tests with users. It will for example be difficult to check automatically if the class and method names fit to the language of the domain where the API is used. An automated measure should nevertheless be able to identify a certain percentage of the usability problems existing within a API, increasing the probability that the rest of the usability problems is discovered with subsequent usability tests and inspections.

1.4 Goals of the Thesis

It is the goal of the thesis to define a measurement method as described above, to show how well it is able to measure usability, and to identify its limitations. The specific goals are as follows:

1.4.1 Identification of Factors Influencing API Usability

To be able to define a method that measures usability effectively, we need to know which factors influence API usability. There are a few existing usability studies identifying such factors [58, 182, 184], but they only cover a very small area of API usability. To gain the necessary insight from a user's point of view, further usability studies must be conducted.

There are three challenges for this goal: The first is to design usability studies that provide meaningful data from which significant results can be derived. This is a crucial step, since conducting any kind of study and interpreting the results takes a lot of time, and one of the worst things that can happen is when afterwards it is discovered that the data is inconclusive. To prevent this, it needs to be defined in detail which questions should be answered by the study, and the study needs to be defined according to them. The second challenge is interpreting the data correctly and finding the factors that influence usability, which can be especially difficult for factors that have not been expected. This process will be supported by a fine-grained and exact evaluation of the data, using video evaluation and appropriate analysis tools. Also, a statistical evaluation of the data will show in which cases significant differences exist between two or more APIs, and factors can be identified based on that. The third challenge will be assigning suitable complexity values to the factors, and bringing the factors in correlation to each other. For example, the measurement method needs to be able to correctly determine if instantiating a certain class is more complex than calling a certain static method. Only then a comparison of different APIs is possible.

See section 1.5 for a more detailed explanation of usability studies and statistical research methods.

1.4.2 Definition of an Automated Framework for Measuring API Usability

With the approach described in section 1.3 as a basis, a process needs to be defined for measuring API usability, based on the given inputs. It needs to be defined which elements of the API and the implemented solution are rated, how they are identified, and how the complexity for each element is calculated. A number of different sources are available for gaining a deeper understanding of API usability, like API design guidelines [45,188], reports about API usability problems [5,94,157], as well as few available API usability studies [58,182,184]. In [13] several characteristics of API usability have been identified, which can be used as a starting point.

The main challenge in this context is how the most important factor concerning usability can be integrated into an automated measurement method: the user. The user's experience and personality can have a significant impact on how well he/she understands an API. It needs to be researched how different aspects of user experience impact usability: the user's overall programming experience, the experience in the problem domain, as well as the experience with the API in question. Usability studies will be used for getting answers to these questions.

It is important to say that the goal of this thesis is not to define a complete measurement method that already produces reliable results for all kinds of APIs. This would hardly be possible, because there are a lot factors that potentially influence API usability, which would take a long time to evaluate with usability studies, and there are likely also factors that have not yet been discovered. The goal is rather to show for selected factors and APIs that the approach is feasible, and to design an extensible API usability measurement *framework*, into which usability-related factors are integrated, and new factors can easily be added later, allowing a gradual improvement of the framework.

1.4.3 Proof of the Integrity and Suitability of the Framework

It needs to be proven whether the created framework is really suitable to measure usability in different scenarios, and to compare the usability of different APIs correctly. The related challenge is to ensure that the information gathered from the usability studies is enough to cover a broader variety of cases. This can be done by a detailed investigation of possible cases prior to the usability studies, an by ensuring that the usability study is designed so that data gathered from it can be generalized to a larger number of use cases.

The suitability of the framework can be shown comparing the calculated results with the results from usability studies, and calculating the correlation between them. A high correlation means that the framework provides suitable results.

To evaluate integrity, a popular method among existing software complexity measures is to use *Weyuker's properties* [198]. We will also use it to evaluate the integrity of our framework, and investigate how well Weyuker's properties are able to deal with a measurement method that, in contrast to existing measures, takes the context of use into account.

1.4.4 Identification of the Limitations of the Framework

It is important to know the limitations of the framework, since it shows which kind of usability-related problem can be found with it, and which ones cannot. A probable limitation will be the naming of API elements, since it will be difficult to check with an automated method if the name is easy to understand and fits to the language of the problem domain. Again, limitations can be identified when comparing the results calculated by the measurement method with the results from the usability studies. In cases where there is a bad correlation, the reason for that must be identified. Ideas for eliminating these limitations will be collected for future work.

1.4.5 Evaluation of a Middleware API Optimized for Usability

Finally, we plan a first application of the usability measurement framework to evaluate a middleware API that was developed by our research group, and is optimized for usability. Using the framework, the API will be compared to other existing APIs in different usage scenarios.

1.5 Methodology

The three main research methods that are used in this thesis are:

1.5.1 Literature Review

A lot of work has already been done in the areas of usability and software complexity measures. A literature review is conducted to identify important related work, and check whether similar ideas have already been published. The related work is also used to roughly identify possible aspects related to API usability, and these aspects are then systematically mapped to possible measurable properties. The properties will then be evaluated using usability studies.

The related work is presented in chapter 2.

1.5.2 Usability Studies

Usability Studies are used in this thesis for identifying the factors influencing API usability, evaluating the influence of user experience, showing the suitability of the created usability measurement method and identifying limitations.

The studies are conducted with APIs that are especially built to look into certain aspects of API usability. They follow the “thinking aloud” method [142], which gives a maximum amount of insight with few required test users. All user sessions are recorded with a screen capturing software to enable a detailed evaluation. From this data, the performance and error rate of the users is extracted, which are two main usability indicators [142], that are also used in other existing usability studies [58, 182, 184]. In addition, the user’s opinions are recorded with a questionnaire, bringing valuable additions to the data gathered from thinking aloud.

Further, eye tracking [149] is used to get a better understanding how users read code and documentation. Gaze plots and heat maps can for example be used to check where a user looked first in a tutorial, or how long he/she took to read a certain text passage.

A detailed description of the study preperation and execution is given in chapters 4 and 5.

1.5.3 Statistical Evaluation Methods

Statistical evaluation methods are used to extract significant data from the usability study results. They help to show whether a potential usability factor has significant influence or not, and how big that influence is. The statistical computing tool R [68] is used for all statistical plots and calculations throughout this thesis. The following statistical methods are used [64, 92]:

Boxplots: Boxplots allow a very easy to understand graphical representation of the distribution of a set of values. Even before applying other statistical methods, boxplots already give a good hint where significant differences could be found between two sets of data.

t-test: A t-test is a parametric statistical hypothesis test for assessing whether one of two samples of independent observations tends to have larger values than the other. In this thesis, it is used to check whether the data from two different APIs are significantly different, helping to identify the factors influencing usability.

Wilcoxon rank sum test [200]: This test has the same purpose as the t-test, but is non-parametric, meaning it can be used when the data is not normally distributed, in which case the t-test is not usable. Nevertheless, the t-test is still used by default because of its stronger significance.

Contingency tables: When binary values are compared (e.g. whether a user finished a task or not), contingency tables can be used for statistical evaluation. The chi^2*-test* is well known for this purpose. For tests with only few users *Fisher's exact test* [66] is especially well suited.

Correlation: Correlation analysis is used to find out whether two variables have an influence on each other. For this purpose, *pearson's correlation coefficient* can be used, as well as *scatter plots* for visual confirmation. Correlation is important for validity checks, e.g. whether the calculated values correlate with the measured ones, or if a developer's experience has any influence on usability.

The statistical methods and their application to the usability studies are explained in more detail in chapters 4 and 5.

1.6 Structure of the Thesis

The thesis is structured as follows: Chapter 2 summarizes related work in the areas of usability measurement and software metrics, and defines important terms like usability. Chapter 3 explains the basic approach for the API usability measurement method. Chapters 4 and 5 describe the two usability studies that were carried out and show which factors were found that influence the usability of APIs. After that, chapter 6 introduces the complete measurement method, based on the results of the usability studies. Chapter 7 evaluates the measurement method and shows how well it can measure usability in comparison to the results from user studies. Chapter 8 applies the measurement method in the area of distributed middleware and presents an API optimized for usability. Finally, chapter 9 concludes the thesis and lists further research issues for future work.

CHAPTER 2

Related Work

The work related to the measurement of API usability splits into two large categories: One is evaluation and measurement of usability, the other is automated measurement of software complexity. One goal of the related work analysis is to show that in the former category there are no measurement methods that are automated and specialized to APIs, and in the latter there are no methods suitable for measuring usability.

Further, in this chapter the most important terms for this thesis are defined.

2.1 Definition of Terms

2.1.1 Application Programming Interface (API)

In the context of this thesis, the definition of the term API is important to be able to specify which kinds of APIs can be evaluated with the automated API usability measurement method. Unfortunately, no scientific literature could be found that deals with the definition of the term API and its characteristics. The Oxford Dictionary defines API as "a set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application, or other service"[1]. This definition shows that there exists a variety of APIs for different purposes:

- **Operating system APIs**: The lowest-level APIs allow communication with the underlying operating system, and are used in low-level programming languages like C. Such an API consists of a number of functions that are offered by the operating system e.g. to control underlying hardware. An example is the Win32 API for Windows.
- **Application APIs**: These APIs allow different applications or components of an application to communicate with each other. They are written in a certain programming language and can normally be accessed with the same programming language. These kind

[1]http://www.oxforddictionaries.com/definition/english/API

of APIs are mostly used in object-oriented programming languages like .Net or Java, that allow component based development, for the purpose of component interaction. Further, these programming languages provide extensive APIs themselves (also called frameworks), which also fall under this category. Such APIs can use the full potential of the underlying programming language, like class inheritance, interfaces and annotations.

- **Service APIs**: These APIs allow access to a certain kind of service, like a web service. They are normally independent of the programming language in which the application that is running the service is written, and the service logic is hosted on a remote server. Good examples are the APIs of popular web sites like Facebook and Twitter. The structure of such APIs is mostly limited to providing functions that accept and provide structured data.

In this thesis we focus on application APIs, especially for the programming languages .Net (C#) and Java. These two languages cover about 60% of the requirements for current programming job offers [122], with the closest competitor being PHP with 15%. We can therefore say that these are two of the most widely used programming languages up to date.

Application APIs can further be distinguished into three different types, which will all be covered by this thesis:

- APIs offered directly by the programming language / framework, e.g. for reading and writing files.
- APIs of external software libraries, e.g. for logging, database access, unit testing or dependency injection. Software repositories like NuGet show that there is a large and steadily growing number of such APIs available.
- APIs of software components, which are part of an application that has been built with the principles of component based software development (CBSD [25, 185]). [185] defines software components as “binary units of independent production, acquisition and deployment that interact to form a functioning system”, and “units of composition with contractually specified interfaces”. These interfaces are the components’ APIs, and define how the components interact with each other. Although in many cases a component’s API will only be used within a single software system, usability is still important, because the API creators and API users are often different programmers. Both Java and .Net are languages that allow and support component based development, with a single component being represented by a single JAR or DLL file.

2.1.2 Usability

There is a number of different definitions for the term usability available in literature. A definition suitable for this thesis needs to be checked for its validity in the context of APIs. Therefore different definitions are compared.

What is Usability

Back when software producers first started to take the user into account, the term of choice was “user friendly” [142]. However, this term is not really appropriate because of several reasons:

It suggests that a system needs to be “friendly” to the user, while it is actually much more important that the system doesn’t get in the way of the users and lets them finish tasks quickly (a good example is Clippy, the “friendly” Microsoft Office help assistant, a feature that drew strong negative response from the users [88]). Further, it implies that the user’s needs can be described by a single dimension of “friendliness”, where in reality different users have different needs. For these reasons, another term has emerged and is now used most often, which is “usability”.

The following definitions for usability can be found in literature (there are others, but they are either outdated, or very similar to the ones given here):

- “The capability of the software product to be understood learned, used and attractive to the user, when used under specified conditions.” [103]
- “The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.” [4, 102, 104]
- “The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component.” [101]
- “Usability means that the people who use the product can do so quickly and easily to accomplish their own tasks. This definition rests on four points: (1) Usability means focusing on users; (2) people use products to be productive; (3) users are busy people trying to accomplish tasks; and (4) users decide when a product is easy to use.” [55]
- Some sources like [142] don’t define usability directly, but say that it is defined by its sub characteristics. A comparison of different definitions of usability sub characteristics is given below.

While there are some differences in the definitions, there are several points that all of them have in common. All definitions contain (1) a product, (2) a user that uses this product, (3) a context of use (analog to “under specified conditions” or ”to accomplish tasks”).

From the given definitions, only the first one has been adapted and explicitly used in the area of APIs [13]. We also adapt this definition for our thesis. Since the product in our case is an API, we define usability as: “The capability of the API to be understood, learned, used and attractive to the user, when used under specified conditions.” Note that this is only a general definition that doesn’t take specific usability sub characteristics into account – more information on sub characteristics is given below.

Who are the Users

Since the user is a central part of the usability definition, it is important to know who the users of an API are. Combining the definitions from [104] and [13], we define a user as a “software developer that uses the API to perform a specific task when developing a software system”, whom we call in short “system developer”. It is important to note that this definition doesn’t include the software developer(s) that created the API, which we further call “API developer(s)”. Also it doesn’t include the actual end users who are using the system that is implemented by the system developers. Figure 2.1 shows how these different roles are related to each other, with the important parts (system developer and API) marked in red.

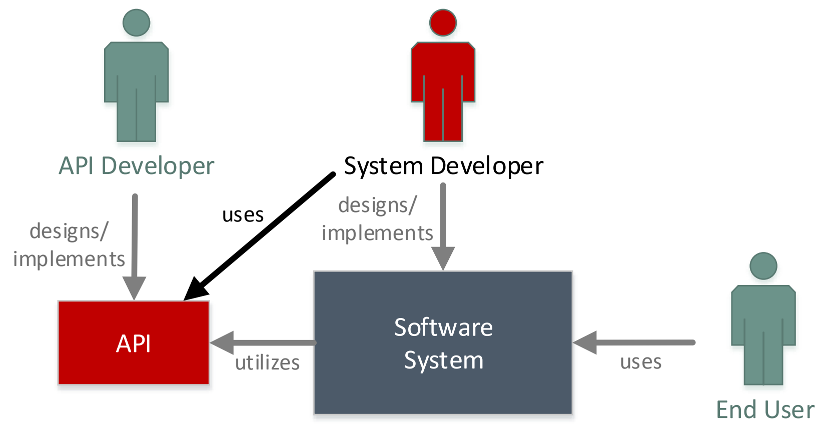


Figure 2.1: Relation of different roles to each other and the API / software system.

Table 2.1: Aspects of the user that influence usability

Skills and Knowledge	**Personal Attributes**
Product experience	*Age*
Task experience	Gender
Organizational experience	*Physical capabilities*
Training	*Physical limitations and disabilities*
Keyboard & input skills	Intellectual ability
Linguistic ability	Attitude
General knowledge	Motivation

To understand how the user is related to usability, the aspects of the user that influence usability need to be identified. Several aspects are shown in [16]. A shortened list of these aspects is shown in Table 2.1. Aspects that are less relevant in the context of APIs are shown in italic. This concerns many of the personal attributes: Age is of minor relevance since developers normally use APIs as part of their work, which means APIs are used neither by children nor by elderly people. Also, physical capabilities and limitations play a minor role since a developer is only sitting in front of the computer. Some deficiencies like bad eyesight may cause problems, but these are related to operating a computer in general, not to APIs specifically. Concerning skills, it can be assumed that a developer has a certain level of knowledge and experience, specifically including skills for operating a computer. Further, the linguistic ability of developers plays only a minor role, since the vast majority of APIs use the English language, and it can be assumed that most developers possess a certain level of skill in this language.

This leaves the developer's knowledge and experience as a main factor influencing usability. This result also coincides with [142], where three main dimensions are identified among which users' experience differs, called the "user cube": the experience with the system, with computers in general, and with the task domain. This is equal to the aspects product experience, general

Table 2.2: Environmental factors that influence usability

Organisational	Job design	flexibility, pacing, autonomy, discretion
	Job structure	work hours, practices, communication structure
	Attitudes/culture	organizational aims, industrial relations
Technical	Equipment	hardware, software, materials
Physical	Workplace conditions	auditory, thermal and visual environment
	Workplace design	space, furniture, user posture, location
	Workplace safety	health hazards, protective equipment

knowledge / training, and task experience from Table 2.1.

The gender of a developer probably has only a minor influence, since typically no gender-specific tasks are done with APIs. Unfortunately there exists no research concerning how an API is approached depending on the gender, so it may be that differences in gender are similar to differences of personas described by the cognitive dimensions framework [37, 83]. This framework uses three different personas, depending on how a developer approaches an API: *opportunistic*, *systematic* and *pragmatic*. E.g. while a systematic persona will prefer reading the documentation before starting to work with the API, an opportunistic persona will start using the API immediately and explore the API features in the code while working.

What is the Context of Use

According to [104], the context of use is defined as "users, tasks, equipment (hardware, software and materials), and the physical and social environments in which a product is used". A more detailed definition of characteristics is given in [16], especially concerning the task and the environment. Important characteristics of the task are: the task goal, the task duration, the flexibility for reaching the goal, and the risk resulting from errors. Important environmental factors are shown in Table 2.2. Next to the user, we see the task as the main factor influencing usability, since the environmental factors are similar for a large number of developers. While an API may be well suited and easy to use for one task, it can be very difficult to use for another task.

There is one additional factor that is of particular importance for the usability of APIs [169]: Most programmers develop software using modern integrated development environments (IDEs) like Eclipse for Java or Visual Studio for .Net, which provide features to help the programmer to correctly use the API. One of the most important of these features is code completion, which provides the programmer with a list of items (e.g. classes or methods) that he/she might need depending on the current context. For example, if a programmer wants to call a method of a certain class, the code completion window will show all of the class's public fields and methods. So, the usability of an API is also strongly dependent on the way its features are presented by the IDE's code completion mechanism. Although the different IDEs for a single programming language tend to present the API features mostly in the same way [169], there are considerable differences between the IDEs of different programming languages like Java and .Net. Therefore

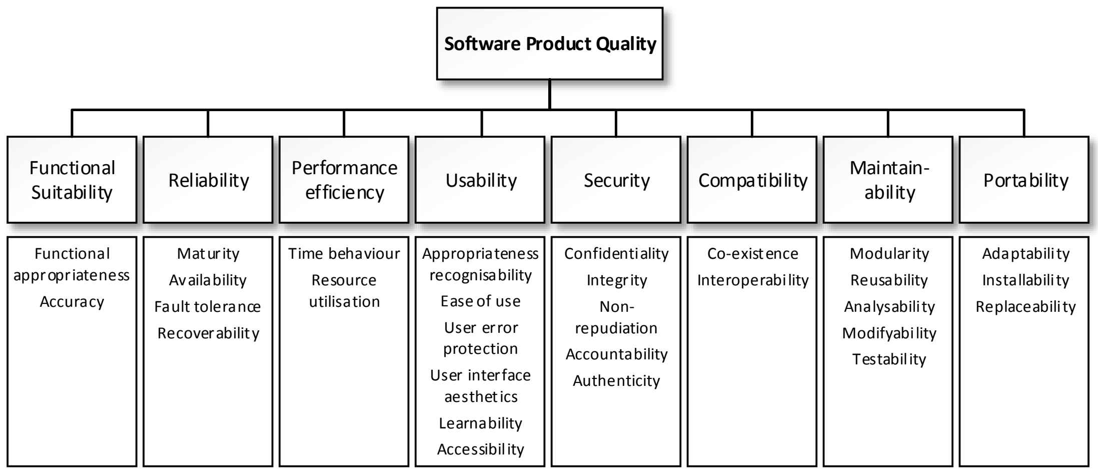


Figure 2.2: The software product quality model [104]

usability may need to be rated differently depending on the IDE and programming language.

Usability in a Software Quality Model

Most definitions of usability are part of a more general software quality model. A large number of different software quality models and attributes are found in literature, many even targeted especially to the area of software components, like [3, 14, 15, 103, 104, 107, 172, 178]. Since it is not in the scope of this thesis, we do not compare the different quality models here, but focus only on a single quality model as an example of how usability is positioned in an overall view of software quality. For this purpose we chose the "software product quality model" that is presented in [104], since it is an ISO standard that also incorporates recent research from this area. Figure 2.2 shows the model's quality attributes and sub characteristics for each attribute. The attributes are defined as follows (usability has already been defined above, therefore it is not mentioned here):

- **Functional suitability**: The degree to which the product provides functions that meet stated and implied needs when the product is used under specified conditions.
- **Reliability**: The degree to which a system or component performs specified functions under specified conditions for a specified period of time. Since aging does not occur in software, reliability problems occur due to faults in requirements, design, implementation and the environment (e.g. hardware errors).
- **Performance efficiency**: The performance relative to the amount of resources used under stated conditions.
- **Security**: The degree of protection of information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them.

- **Compatibility**: The degree to which two or more systems or components can exchange information and/or perform their required functions while sharing the same hardware or software environment.
- **Maintainability**: The degree of effectiveness and efficiency with which the product can be modified. From the perspective of support staff, maintainability can be interpreted as a quality in use for the goal of maintaining the product: the degree to which a product meets needs to maintain the product with effectiveness, efficiency, safety and satisfaction in specific contexts of use.
- **Portability**: The degree to which a system or component can be effectively and efficiently transferred from one hardware, software or other operational or usage environment to another. From the perspective of support staff, portability can be interpreted as usability for the goal of transferring the product from one environment to another.

In [178] an evaluation of quality attributes for software components is presented, where software developers and project managers experienced with component based software development were asked to give an importance rating for each quality attribute. Six quality attributes were rated, equal to the ones shown above, except for security and compatibility. While the top rated properties were portability and reliability, the one that was rated by far the least important was usability. Although this may have changed by the time since this evaluation was done (in 2003), it may explain why there exists so few research on the usability of software components / APIs.

Usability Sub Characteristics

More important for this thesis than general quality characteristics are the sub characteristics of usability. They are an important part of the definition of usability, and so they of course need to be taken into account when usability is evaluated. There are several different definitions of usability sub characteristics in literature. A comparison of these characteristics is shown in Table 2.3. A comparison of additional characteristics can be found in [175], but most of them are not included here since they are outdated and/or can be completely mapped to those shown in Table 2.3. Characteristics that are equivalent are shown next to each other in a row. Whether a characteristic is equivalent to another one has been deducted from their detailed descriptions in the respective sources. [13] maps the characteristics from [103] especially to software components, but identifies no particular differences for that area.

In this thesis we mainly use the names of the characteristics presented in [104], since it is the most up to date model, has integrated the characteristics of older models, and has renamed some of them for better understanding. The following list gives more detailed explanations for the characteristics:

- **Ease of use**: The degree to which users find the product easy to operate and control. This includes the controllability of the product, and how well the product conforms to the user expectations. This characteristic is rather difficult to map among the different models, as "efficiency" is defined similar but not completely the same. [142] emphasizes that it is especially important how well the product can be used by expert users, in other words the performance at the time when the learning curve flattens out.

Table 2.3: Comparison of usability sub characteristics of various models

Constantine et al. [40]	**Nielsen [142]**	**ISO 9126 [103]**	**ISO 25010 [104]**
Efficiency in use	Efficiency	Operability	Ease of use
Learnability	Learnability	Learnability	Learnability
Rememberability	Memorability		
Reliability in use	Errors/safety		User error protection
User Satisfaction	Satisfaction	Attractiveness	User interface aesthetics
		Understandability	Appropriateness recognizability
		Usability compliance	
			Accessibility

- **Learnability**: The degree to which the product enables users to learn its application. The fact alone that this characteristic is represented in the same way in all models shows that it is very important. [142] calls it the most fundamental usability attribute, since "the first experience most people have with a new system is that of learning how to use it". A typical representation of learnability is a learning curve. Systems that are highly learnable allow users to reach a high level of proficiency within a short time. The learning curve can be strongly influenced by the experience of the user – a user may learn much faster if he/she can transfer skills e.g. from a previously learned product. When analyzing learnability, it should be kept in mind that users normally do not learn by completely reading a documentation or tutorial before using a product for the first time. On the contrary, users often start using the product without much preparation, and learn by exploration [142], which makes e.g. easy to understand error messages very important.
- **Memorability**: The degree to which a user can memorize how to use a product. This characteristic is not found in the ISO standards. An explanation for that may be that memorability potentially correlates with learnability – the easier it is to learn something, the easier it is also to memorize it. Memorability is especially important for casual users, in other words people that are using a product infrequently. This can also concern parts of a product that only need to be used in exceptional circumstances.
- **User error protection**: The degree to which the system protects users against making errors. An error is any action that does not accomplish the desired goal. Errors can be very different in severity: Some errors are recognized and corrected immediately by the user and have no other effect than to slow the user down, and are only of minor relevance. Much more important are errors that are not (or cannot be) discovered by the user, leading to a faulty task outcome. Such errors are difficult to recover, and are therefore the main concern of this characteristic.
- **User interface aesthetics**: The degree to which the user interface enables pleasing and satisfying interaction for the user. For graphical user interfaces this concerns aspects of

the design like colors and fonts. This is especially important for software that is used in a non-work environment, e.g. games, since in this case users want to have an entertaining experience [142]. This makes it apparently less relevant for APIs, since they have no graphical interface at all. There exists no research concerning what factors influence satisfaction in case of an API.

- **Appropriateness recognizability**: The degree to which the product provides information that enables users to recognise whether it is appropriate for their needs. A software is approriate if it is suitable for the specific tasks and user objectives that need to be accomplished. Especially important for this characteristic is the information that can be accessed without actually acquiring the product, including tutorials, documentation, and a product web site. This is critical, because even when a product is easy to use, it cannot attract users when they are unable to find out if using it for a certain task is appropriate.
- **Usability compliance**: The capability of the software to adhere to standards, conventions, style guides or regulations relating to Usability. We see this characteristic of only minor relevance, because of two reasons: First, it only occurs in a single quality model and has not been adopted into the most recent ISO standard [104]. Second, while it is good to adhere to guidelines, [16] points out some problems and limitations: Detailed and specific guidelines are likely to be appropriate only for specific systems and types of users. On the other hand, very general guidelines are difficult to interpret. Following guidelines does not ensure that a product reaches any particular level of usability – it is much more important to consider whether a specific task can be carried out better with a specific design.
- **Accessibility**: Usability and safety for users with specified disabilities, e.g. associated with age. As it has already been analyzed, this aspect is only of minor relevance in the context of APIs, since no young or elderly people are using them, and an API can be used by any person that is able to operate a computer.

2.2 Evaluation and Measurement of Usability

Figure 2.3 shows a categorization of related work concerning the evaluation and measurement of usability. Standards and Definitions have already been examined in section 2.1. All other categories are further described here. The category marked in red is the one that this thesis falls into.

2.2.1 Usability Measurement

The currently available methods for measuring usability can be divided into three different categories [67], which are user testing, usability inspection and usability inquiry. The following sub sections explain existing usability measurement methods within these categories in more detail.

User Testing

This approach requires representative users to work on several tasks with the system under test. Testing is possible with prototypes, but enough functionality is needed to convey the impression

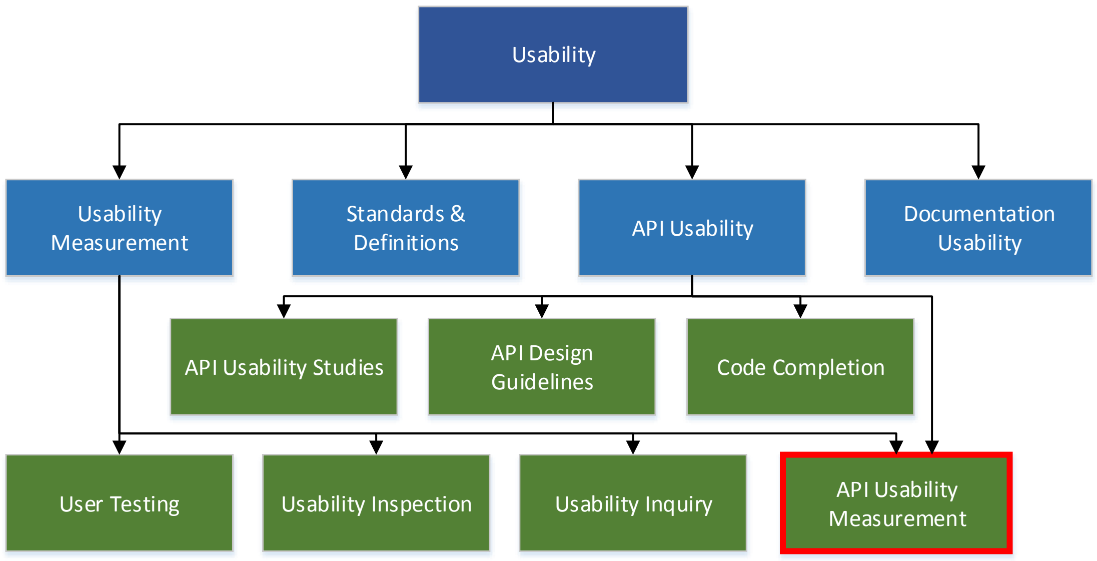


Figure 2.3: Categorization of Related Work for Usability

of a working system to the users. Evaluators use the results to see how well users were able to solve the tasks, and where they had problems. Next to the users' opinions, typically collected quantifiable results are [7, 142]:

- The time users take to complete a specific task
- The number of tasks that can be completed within a given time limit
- The number of successful/failed tasks
- The number of errors and recovery time
- The number of times the user expressed clear frustration or joy

The most important advantage of test methods following the user testing approach is the direct involvement of the user. Having real users allows identifying usability problems that occur in real-world use, while other measurement approaches often only provide guesses where problems could be. While the guesses may often be good, they have a greater chance to be wrong or just incomplete when no real users are involved. The disadvantages are that a lot of effort is required since a representative set of test users needs to be found, a set of tasks needs to be prepared, tests need to be recorded and results evaluated. Further, a product is required where at least the functionality that should be tested needs to be implemented already to a certain degree. While it is possible to test with earlier prototypes (e.g. paper prototypes [174]), the significance of the results suffers because users may be influenced by the abstractness of the prototype.

A widely known test method is *thinking aloud*, about which [142] says that it "may be the single most valuable usability engineering method". It involves having a test user thinking out loud while doing something, allowing the evaluators to get a better understand of the user's actions, and identify major misconceptions. Several variations of thinking aloud have been introduced [55, 142]: *Constructive interaction*, also called *co-discovery learning*, lets two users work together, with the advantage that users can more easily verbalize what they are thinking when talking to each other. A disadvantage is that twice as many users are required, and that sometimes two users are paired that have problems working together. With *retrospective testing*, a video recording of the test session is reviewed together with the user, allowing to get more detailed information from the comments of the user while reviewing the video. This method can be beneficial when it is difficult to get a representative set of test users, since it becomes possible to gain more information from each test user. The *coaching method* is another variation, which differs from other usability tests in having an explicit interaction between the test user and the evaluator (or "coach"). It is aimed at discovering the users' information needs e.g. to provide better training and documentation, and if possible redesign the interface to avoid the need for the questions. In another variation, called the *teaching method* [189], a test participant that has become familiar with the system demonstrates it to a seemingly naive user (actually a confederate of the evaluator) and describes how to accomplish certain tasks. Compared to the traditional thinking aloud method, this has the advantage that users tend to be more talkative and therefore a better insight can be gained on their thought processes.

One problem with user testing is bringing users and evaluators together (same time and space), which can in some cases require increased effort. A way to avoid this is *remote evaluation* [91], in which case users and evaluators are seperated. This can either mean that they are remotely connected during the test (space independent), or even that the test is being recorded without the evaluator (time and space independent), and the data is later being transferred to the evaluator for interpretation. Although this requires a more complex software/hardware setup, it increases flexibility and reduces effort, and has the additional advantage that test users could even do the test directly from their work place.

A special kind of user test is the *A/B test* [87]. This test allows comparing two or more variants of a user interface and rate each variant objectively. This is done by implementing all of the user interface variants, doing the same user tests with all of them and measuring performance values like the ones mentioned above. Such a test provides very accurate and also statistically analyzable results, but to get these results a lot of effort is needed, since multiple user interfaces need to be designed and tested, and a larger number of users is required.

Another user testing approach that should be mentioned is *eye tracking* [149], where special hardware is used to track the user's eye movements. The resulting data can be evaluated in different ways, like heat maps and gaze replays. Several new usability evaluation approaches are starting to make use of eye tracking, like the *total-effort metrics* [61, 186, 187], which combine it with other interesting data like recorded key strokes. While eye tracking allows getting results that cannot be acquired with any other method, it requires a lot of effort for setting up the tests with the expensive eye tracking hardware and for evaluating the resulting data. Also, experts are needed to do proper eye tracking studies because there are many details that need to be considered and data can easily be misinterpreted.

Finally, a less effective but very simple method is *hallway testing* [21]. In this case the programmer simply asks about 5 random people (met coincidentally on the hallway, therefore the name) to have a look at the software or design in question. With this method a majority of the usability problems can reportedly be found, so although it may not reveal all problems, it is a good idea to do it prior to a “real” usability study.

Usability Inspection

This approach requires usability specialists and other professionals to examine a user interface and judge whether its elements follow certain usability principles. Usability inspection methods can generally be split into ones that evaluate specific scenarios (generally called “walkthroughs” [82]), and ones that don’t (e.g. guidelines). An overview of usability inspection methods can be found in [98].

Advantages are that no users are needed which significantly reduces the testing effort, and that the user interface can be evaluated without any (or only dummy) functionality being implemented. Disadvantages are that the approach is less reliable for finding usability problems because of the missing user interaction, and highly depends on the experience of the evaluators. Also, to find a majority of the usability problems, multiple evaluators are needed (e.g. at least 5 evaluators when 75% of the problems should be found) [141]. This is also necessary to minimize the “evaluator effect” [96], which shows that there can be large differences between the usability problems that are found by different evaluators. This effect can be as strong as two different evaluators agreeing in only 5% of the found usability problems.

The most basic inspection method is *standards/guidelines inspection* [202], which is a systematic inspection of a user interface design, where evaluators check if certain predefined guidelines are fulfilled. This method suffers from the problem that guideline lists tend to be very large (up to thousands of items) and therefore hard to apply. Because of this problem, another method emerged, which is called *heuristic evaluation*. Heuristics are very similar to guidelines, but usually more general and fewer in number (a single set of heuristics consists of about 10 items). Different sets of heuristics have been introduced in literature, e.g. [77, 142]. How efficiently usability problems can be found therefore not only depends on the evaluator’s experience, but also on which set of heuristics is used. Concerning the usability of APIs, a problem is that existing heuristics are not very well suited for evaluating APIs because they are often targeted to graphical user interfaces. For example, [142] defines the heuristics “simple and natural dialogue”, “feedback” and “consistency”. While the latter is completely applicable to APIs (consistent naming of classes, methods, ...), the other two are not, since the API simply doesn’t provide a dialogue or direct feedback while writing the code. There is only a single known usability study where a set of heuristics was applied to an API [85]. There a set of 16 heuristics was used which the authors derived from a report of usability problems [210]. Unfortunately, there are no other known reports where these heuristics have been used and/or evaluated.

Several variations of heuristic evaluation have been introduced, one being *perspective-based usability inspection* [208, 209], where each inspection session focuses on a subset of usability issues covered by one of several usability perspectives, instead of evaluating all issues at once. This allows a better focus of the evaluator’s attention, and makes it easier to apply the method also with less experienced evaluators. Three different perspectives are used, which are “novice use”, “expert use” and “error handling”. A comparison to heuristic evaluation showed that the method tended to produce better results with the same number of inspection sessions (though suffering from very low statistical significance) [209].

Another inspection method is *cognitive walkthrough* (CW) [124, 152, 199], which focuses on evaluating the learnability of a user interface by applying cognitive theory. It is especially appropriate when users need to learn how to use an application through exploration. In contrast to heuristic evaluation, CW is task-based, meaning the interface is evaluated in the context of specific user tasks. Performing a walkthrough involves carrying out a simulation of the cognitive processes that are required to successfully guess the sequence of actions for a certain task. Therefore the evaluator needs to consider things like a user's background knowledge that influence cognitive processes when he/she uses the interface. While CW can provide a deeper insight than heuristic evaluation especially from the viewpoint of specific users, [124] shows that it tends to find less usability problems and require more effort than heuristic evaluation (though still less than user tests). It also requires highly experienced evaluators that are able to correctly guess the users' thought processes.

Another method that builds on cognitive theory is the *cognitive dimensions framework* [19, 83, 84]. It defines several criteria, called "cognitive dimensions", that aim to give designers a common vocabulary, and therefore a common way to think about designed artifacts. They are therefore in some way similar to heuristic evaluation, except that their primary source is the cognitive domain. They are not only applicable to software products, but generally to any kind of "information artifact". Examples for cognitive dimensions are *viscosity* (resistance to change) and *premature commitment* (constraints on the order of doing things). What is especially unique about this method is how the characteristics of the user are taken into account. This is done with user profiles, where the importance of each cognitive dimension is rated for a certain type of user. The user profile can then be compared with the measurement results by looking how well the values for each cognitive dimension overlap. This allows comparing different user interfaces in a way that is impossible with other usability inspection methods. There are, however, also problems with the cognitive dimensions framework [65]: Although the method is said to be targeting novice users, the cognitive dimensions are not easy to understand. Some dimensions are closely related to each other, so changing a user interface concerning one dimension will most likely also influence other dimensions, making it difficult to know how to improve a user interface. Further, novice users are having problems giving an appropriate rating for each dimension, since experience is needed to know which situations typify a high or low rating.

What is special about the cognitive dimensions framework concerning the scope of this thesis is, that it has been explicitly ported for evaluating APIs [37, 39]. It is for example used by Microsoft to evaluate the APIs of the .Net Framework. For this purpose several of the dimensions have been altered especially for the context of APIs, e.g. *progressive evaluation* (to what extent can partially completed code be executed to obtain feedback on code behavior?). Three different developer personas are defined (systematic, pragmatic and opportunistic) and mapped to the different dimensions. E.g. while progressive evaluation is important for opportunistic developers that learn by exploring the API, it is unimportant for systematic ones. Since the dimensions are defined in a more concrete way for the area of APIs, they are easier to understand, but the problems mentioned above still remain to a certain degree.

Yet another usability inspection method similar to CW is the *pluralistic usability walkthrough* [17]. In addition to usability experts, it incorporates representative users and product developers. All of them are asked to assume the role of the user. For a certain task, the application's screens are presented in the same order as they would appear to the end user, and participants write down what actions they, as users, would take for each screen. After that the group discusses each screen, with the representative users speaking first. Advantages are that the perspective of the user is better represented since there are actual users involved, and that feedback can be obtained even if the interface is not fully developed. On the other hand, the approach is limited due to its representation as a sequence of screens: Only representative (rather than comprehensive) user paths can be evaluated. In the area of APIs, the pluralistic walkthrough cannot be directly used because there are no actual "screens" that can be presented. But a very similar method that is targeted to APIs has been introduced in [60], called *API usability peer reviews*. With this method, the developers of an API are brought together with a group of potential API users. Under the guidance of a moderator/evaluator they walk through a number of code examples. As with pluralistic walkthroughs, the developers benefit from getting direct feedback from the users and being able to discuss issues with them directly. A study presented in [60] shows that while an API peer review doesn't find as many bugs as a traditional user test, it is over all more time-efficient (i.e. it finds more bugs per unit of time).

Several studies (e.g. [105, 110]) have been conducted that compare inspection methods with empirical usability studies (user testing). They concluded that heuristic evaluation found an equal or even greater number of usability problems than user tests, though the latter revealed more severe problems, more recurring problems and more global problems than heuristic evaluation. Further, inspection methods produced a higher number of possible "false alarms", issues that may never bother users in actual use. But in general, the two methods are also complementary since they have different sensitivity to different types of issues [60]: While user tests expose design issues related to actually using a software product and related resources including documentation, inspection methods tend to expose the design rationale and uncover conceptual design flaws. A good example are tests that have been conducted for the .Net Framework APIs for reading and writing files [38], using the cognitive dimensions framework: While user tests suggested that users had problems because of insufficient documentation, an inspection of the APIs identified problems with the overall design (the required classes had the wrong abstraction level).

Usability Inquiry

This approach, also known as "field usability research", requires evaluators to obtain information from users about what they like and dislike, as well as their needs and understanding of a system. This is done by e.g. asking them questions (via interviews or questionnaires) or observing them while using a system in real work. Methods following this approach differ from user tests and usability inspection in two ways: First, they do not require that a design or prototype is produced before data can be collected. This means that they can influence design in early stages. Second, they offer ways to analyze how the user currently works, so that a new product can be based on actual user needs.

Although usability inquiry methods are widely used, they are not used as frequently as usability tests/inspections [203]. The main advantage of these methods is that real users are analyzed in a real environment doing real work. But it is also their greatest challenge. Field methods often take a lot of time and effort and produce an overwhelming amount of data, which must be recorded and analyzed correctly to capture detailed and unprejudiced results. Further, finding users that are willing to be observed/questioned can be complicated, even more if they are working in companies with strict policies. In case of APIs, observation can be even harder, since developers normally don't work with a single API on a daily basis, but rather use it infrequently as only one of many development tools. So it may be very inefficient for an observer to sit next to a developer, only to watch him/her use the API sporadically, and do a lot of other stuff in between. Also, if the usability of the software itself should be inquired (and not only requirements data should be collected), it needs to be already finished and in use.

In [142] three different usability inquiry methods are presented: *Field observation* is the act of observing one or more users using the interface in their work environment. An evaluator observes the user, letting him/her work as undisturbed as possible to not falsify the results. Compared to user tests, the method has the big advantage that users may be observed using the software in ways that were not expected, doing things that would not have been tested during a usability test. In *focus groups*, about six to nine users are brought together to discuss new ideas and identify existing issues. A moderator is responsible for maintaining the focus of the group on the topics of interest, while the session should feel free-flowing for the users, which allows bringing out ideas in an open and dynamic group discussion. This means that one of the most important things for this method is an experienced moderator. Further, it is important to have a representative number of users, and preferably multiple focus groups should be run to get representative results. Another inquiry method is *logging actual use*, which involves having the computer automatically collect statistics about the use of a system. It is particularly useful because it shows how users actually work, and it is easy to automatically collect data from a large number of users, to see e.g. which features have been used or the frequency of specific events like error messages. The resulting statistics can be used to optimize frequently used features, or identify ones that are rarely used. Logging is usually achieved by either instrumenting low-level parts of the system (e.g. keyboard and mouse input), or by modifying the software of interest. So, although no evaluators are needed for the actual observation, enabling logging can become quite a difficult task that requires technical expertise and access to system resources that company security policies may prevent. Adding logging functionality is possible with APIs – but for logging what the user actually does, not the API itself would need to be observed but rather the used IDE. Since the developer does many other things with the IDE in addition to using the API, logging API usage is a rather difficult task – no existing research has been found on this topic. Logging can also be used as a supplementary method during user testing to collect more detailed data.

One inquiry method that has been introduced particularly for the area of APIs is the *concept maps method* [78]. With this method, developers using an API are asked to draw a concept map visualizing the interactions of their software with the API, and annotating API elements with their impressions of them (e.g. whether an API element was easy or frustrating to use). Several sessions are done in succession, e.g. once a week, where the developers can alter their concept map from the last session. Each session is video-taped and participants are thinking aloud. The resulting data is used to get a deeper understanding of how users conceive an API and how this conception changes over time (e.g. an API element may seem complex in the beginning, but the developer may come to recognize its advantages after having used it for some time). This is especially interesting because it cannot be evaluated with normal user tests that only last for one or two hours at best. As with all inquiry methods, a big disadvantage though is that an API can only be evaluated when it is already finished and in use.

Further, different methods have been introduced for conducting usability-related interviews. A well known interviewing method is called *ethnographic interviewing* [106, 109]. It is based on the idea that standard interviewing techniques often overlook shared assumptions, contextual understandings and common knowledge. During the interview, the interviewer tries to bring out these assumptions and understandings. Questions have the goal to reveal perceptions and knowledge that guide the users' behavior, while discouraging them from translating this information into something that seems easier to understand for the reviewer (because significant detail tends to get lost when users try to do that). Another interviewing method is *contextual inquiry* [201, 203], which combines interviewing with observation techniques. It criticizes that with conventional interviewing users are too far away from their work and tend to talk in more abstract terms, and therefore suggests to interview users while observing them at their work. Also, it reverts the relationship between user and interviewer, giving the user a master/teacher role which lets him/her talk more freely and bring out new views and ideas. The method is part of a larger process called *contextual design*, which additionally cares about prototyping and design iterations.

Another very popular inquiry method are questionnaires. They are similar to interviews, with the difference that they require fewer usability personnel, but interviews are more flexible since the interviewer is able to rephrase questions or ask for additional details. Many different usability questionnaires can be found in literature, e.g. [18, 23, 33, 112, 123, 125, 142], providing different levels of complexity, comprehensiveness and focus (e.g. on pragmatic or emotional aspects). One concrete example is the *user experience questionnaire* (UEQ) [123], which is a questionnaire with 26 different item pairs (e.g. confusing/clear, conservative/innovative), rating both emotional and pragmatic aspects. Each pair is rated on a seven point differential scale, and is mapped to one of six different Factors, which are attractiveness, perspicuity, efficiency, dependability, stimulation and novelty. This mapping allows to quickly evaluate the questionnaire results to get ratings for the six factors. While comprehensive questionnaires like UEQ have the advantage of giving a good representation of the overall usability of a product, they are far less suited to find specific usability problems, for which a questionnaire is needed that covers certain areas of the product specifically (e.g. SUMI [112]). Most often questionnaires are used in addition to other usability tests, e.g. with user tests, to record the user's opinion of the product. As with other inspection and inquiry methods, the suitability of a questionnaire for evaluating APIs depends on the defined questions. General questions or items like the ones used in UEQ are also suitable for APIs, but more specific questionnaires may need adaptations.

Comparison of Methods

A comparison of available methods is shown in table 2.4: *Simplicity* refers to how easily the method can be applied – a negative rating implies that the method requires expert evaluators. *Early Evaluation* means whether the method can be applied in early design stages without having a functional product. *Effort* rates how much effort is needed for using the method, including things like time, cost and number of test users. *Effectiveness* rates how well the method can identify usability problems. *API Suitability* rates how suitable the method is for evaluating the usability of APIs. It should be noted that comparing different usability measurement methods, or even evaluating the efficiency of a single usability measurement method, can be a very complex task, and data can easily be misinterpreted. [82] analyzes some of the most popular evaluations of usability measurement methods and shows that there are severe threats to their validity (e.g. missing statistical evaluation, low sample size, wrong conclusions drawn). We therefore point out that, although the comparison of evaluation methods shown here represents the state of related work, it may be partly inaccurate, depending on the validity of the related work.

The comparison in table 2.4 shows that evaluation methods of the same category often have things in common. User tests are most effective, but also require the most effort and don't allow to evaluate interfaces as early as other methods. Usability inspection methods need less effort and allow early evaluation, but are also less effective and require experienced evaluators. Usability Inquiry methods allow evaluating usability requirements even before the design of the product is started, but the product itself can only be evaluated when it is finished. Therefore, the focus is actually a bit different than with the other two categories, and inquiry methods can rather be seen as an addition to the other two than a replacement.

When categorizing the methods by purpose, we can identify three categories: (a) gathering and evaluating usability requirements, (b) evaluation of the user interface during design and production, and (c) evaluation of the finished product. While user tests and inspection methods both fall into category (b), inquiry methods mainly cover categories (a) and (c). The measurement method that is introduced in this thesis targets category (b), we will therefore concentrate on a comparison with user tests and inspection methods.

Concerning the suitability of the methods for APIs, user tests are well suitable because the tests are very generally applicable to any kind of user interface. Inspection methods are less suitable because many rely on guidelines or heuristics that are specific to a certain kind of interface (e.g. graphical user interface). The comparison in table 2.4 shows three methods to be most suitable, having at least two positive (+) and two medium (~) ratings, which are hallway testing, the cognitive dimensions framework and API usability peer reviews.

The advantage of hallway testing is that it is very simple and doesn't require much effort, and allows to identify the most obvious usability problems in a very short time, which are also main goals for our measurement method. Unfortunately there exists no evaluation of the suitability of hallway testing for APIs, so it is unclear whether letting someone take a quick look at a piece of code is as effective in finding usability problems as letting someone take a look at a graphical user interface. Also, especially with one-man open-source projects where there are no other people around, hallway testing may be difficult.

The cognitive dimensions framework and API usability peer reviews have shown to be the only usability evaluation methods that have been explicitly created/adapted for APIs, and are also actively used for this purpose. They are therefore very well suited for measuring API usability. The only drawback of the cognitive dimensions framework is that it is not quite easy to use, since evaluating the dimensions (e.g. giving reasonable ratings, understanding interdependencies) requires some experience. API usability peer reviews on the other hand are easier to understand, but have the drawback of requiring the presence of suitable test users.

The bottom line of table 2.4 shows the goals for the new measurement method that is in-

Table 2.4: Categorization of Usability Measurement methods

Method	**Category**	**Simplicity**	**Early Eval.**	**Effort**	**Effectiveness**	**API Suitability**
Thinking Aloud [142]	User Test	~	-	-	+	+
Constructive Interaction [55, 142]	User Test	~	-	- -	++	+
Retrospective Testing [142]	User Test	~	-	-	+	+
Coaching Method [142]	User Test	~	-	-	+	+
Teaching Method [189]	User Test	~	-	-	++	+
Remote Evaluation [91]	User Test	-	-	~	+	+
A/B Test [87]	User Test	~	-	- -	++	+
Eye Tracking [149]	User Test	-	-	- -	++	~
Hallway Testing [21]	User Test	+	~	+	-	~
Standards Inspection [202]	Inspection	~	+	~	-	-
Heuristic Evaluation [77, 100, 142]	Inspection	~	+	~	~	-
Perspective-Based Usability Inspection [208, 209]	Inspection	~	+	~	~	-
Cognitive Walkthrough [124, 152, 199]	Inspection	-	+	~	~	~
Cognitive Dimensions [19, 83, 84]	Inspection	~	+	~	~	+
Pluralistic Usability Walkthrough [17]	Inspection	~	+	~	~	-
API Usability Peer Reviews [60]	Inspection	~	+	~	~	+
Field Observation [142]	Inquiry	~	~ [1]	~	+	-
Focus Groups [142]	Inquiry	-	~ [1]	-	+	+
Logging Actual Use [142]	Inquiry	~	-	~	+	-
Concept Maps Method for APIs [78]	Inquiry	~	-	-	+	+
Interviews, e.g. [106, 201]	Inquiry	~	~ [1]	~	~	+
Questionnaires, e.g. [33, 112, 123]	Inquiry	~	~ [1]	+	-	~
New Measurement Method	*Calculation*	+	+	+	~	+

1) Early requirements evaluation possible, but actual software can only be evaluated when already in real use

troduced in this thesis. It should be easy to use (no experienced evaluators), require little effort (no test users or complicated test procedure), and of course be especially targeted to APIs. As the method is automated, it requires neither tests with users nor evaluators that do some kind of inspection, and therefore falls into a new category named “calculation”. Being automated and not involving users of course also makes detecting certain kind of problems impossible, so the method will never be as effective as user tests, but the goal is rather to be an addition to traditional tests, and an alternative for people that have no experience and/or cannot put much effort into usability evaluation.

2.2.2 API Usability Reports, Guidelines and Studies

Several articles have pointed out the importance of API usability [5, 20, 37, 94, 133, 157, 163]. Especially interesting for this thesis is which factors influence API usability, and therefore need to be considered in an automated measurement method.

[211] presents an evaluation of a large number of bug posts related to APIs, and tries to relate them to a list of aspects that make APIs difficult to use as identified in [210]. The most difficulties unsurprisingly come from erroneous or missing API features, as well as incorrect or incomplete API documentation. After that, the following aspects were identified to be most influental:

- **Exposure of elements**: Unnecessary exposure of the internal API logic, like internal classes and fields, can confuse the user and therefore lead to bad usability. On the other hand, hiding functionality that may be necessary for more advanced users may also pose problems. Exposed types should be immutable and non-inheritable if using a type in that way is not intended, to prevent any kind of misuse.
- **Memory management**: Memory management (allocation and deallocation of memory) responsibilities that are left to the user reduce API usability. This makes C++ difficult to use compared to Java and .NET, which automatically take care of these issues. For example, the CORBA C++ API requires callers to do all memory management themselves, increasing the danger to make mistakes that lead to memory corruption [94].
- **Function parameters and return values**: The number and types of function parameters and return values have significant impact on usability. Functions with many parameters, especially those with multiple consecutive parameters of the same type, are difficult to use [20].
- **Technical mismatch**: Compatibility with the platform and other technologies in the functional environment is important for usability.
- **Dependency**: Strict dependencies make an API inflexible. Using interfaces loosens dependencies between components and is therefore better than having dependencies on concrete implementations.
- **Backward compatibility**: Missing backward compatibility may lead to usability problems for users upgrading to a new version of an API.

Though this list is of course strongly influenced by the method of evaluation that was used in [211] (e.g. bad naming may often not lead to a bug post), it gives valuable hints about important API usability aspects. A more extensive list of factors that influence the usability of APIs is presented later in chapter 3.

API Usability Studies

So far only few studies can be found in literature where the usability of one or more aspects of an API have been evaluated: [58] analyzes the role of the factory pattern in API design, comparing factory methods to traditional instantiation with a constructor. It comes to the conclusion, that users require significantly more time when using a factory than when using a constructor. In the study a factory was always perceived more complex than a constructor, so using a constructor should always be the preferred solution. This is an interesting result considering the popularity of the factory pattern.

Another study presented in [182] investigates the usability implications of requiring parameters in object's constructors. While it may seem logical that required parameters would create more usable and self-documenting APIs by guiding programmers toward the correct use of objects and preventing errors, the study shows a different result. Users performed much better with a create-set-call approach, where first an object is instantiated with a parameterless constructor and then public setter methods or properties are used. An analysis of the participants' behavior using the cognitive dimensions framework showed that a constructor with required parameters interferes with common learning strategies, causing undesirable premature commitment, and therefore a create-set-call approach should be preferred.

[184] presents a study which analyzes the implications of method placement on API learnability. It shows that programmers are significantly faster when a method is placed in a class they are already using than when it is placed in a separate helper class (e.g. when sending a message, `msg.send()` is easier to use than `Transport.send(msg)`). Not finding an expected method, participants sometimes questioned their (correct) choice of starting class, leading to even more confusion. A good placement of methods can guide the programmer, provide a natural flow through classes and tell him/her what class to use next.

A study published in [150] analyzes a persistance API for the programming language Eiffel. It has only been published as a technical report, and only a single API has been analyzed, so the presented results are to be taken with caution. The report analyzes several potential usability issues, among which are also the ones presented in the other three studies. It confirms problems with the factory pattern and with using methods of classes that are not directly related to each other, but doesn't find evidence that required constructor parameters are a problem. Other results are that the documentation is very important, and also that strings requiring a special structure need to be taken into account (e.g. SQL-like query strings) when evaluating an API's usability.

Finally, [85] presents a study where an API was evaluated that allows adding extensions to a support tool for contextual usability study setups (i.e. a highly specific problem domain). The study was done in three phases, first using heuristic evaluation with 4 expert evaluators, second with a developer workshop, and third with interviews. The used heuristics were created directly by the authors, and derived from a report about usability problems in [210]. The workshop included 8 participants, which first were briefly introduced to the API and the corresponding problem domain, and then had to solve different tasks with it. The problems identified in all three phases were classified according to the used heuristics. The study concludes that user-based tests do not find more problems, but problems of higher severity, than heuristic evaluation, which is in accordance to other studies [105, 110]. Though the results are of limited value, since they were not statistically evaluated. Overall, this study focuses more on comparing different usability evaluation methods for APIs, rather than identifying concrete problem areas of API usability.

Table 2.5 presents details on how these studies were carried out, giving a valuable basis for what to consider when conducting API usability studies. All studies except [85] used statistical evaluation methods, averting most threats to their validity. One threat is that in three studies the users learned how to use an API by exploring it, without any kind of tutorial being available. With a tutorial, problems could have a different impact, e.g. requiring constructor parameters may pose less of a problem when a tutorial example clearly shows how these parameters are filled (the results of the study in [150] also support this assumption). Another threat to the validity is that in four of the studies only a single programming language was used, so it is unclear whether e.g. .Net users might have behaved differently than Java users.

The biggest problem with the existing API usability studies though is, that they only cover a very small area of API design patterns. This is also shown in an analysis of the space of API design decisions in [183]. So, many factors of API usability still remain unclear, making it absolutely necessary to conduct additional studies to be able to use the results as the basis for a usability measurement method.

API Design Guidelines

Since the goal of API design guidelines is to improve the usability of APIs, it needs to be checked whether they can be used as input for the usability measurement method. There are several books about usability guidelines available [45, 188], as well as online sources (e.g. [75]). Although such guidelines give a good basic understanding about how to design an API, their problem is that they are mainly based on the authors' experience, and lack a scientific basis. This becomes especially obvious when comparing guidelines with existing API usability studies. E.g. [75] suggests to always place required parameters in the constructor, while the study in [182] comes to a different conclusion. Because of that, we will use guidelines only as input to find out what may influence the usability of APIs and needs to be checked with a usability study, but not as a direct input for the measurement method.

Special Elements of API Design

When writing code in general, and when using an API in particular, a programmer is not only influenced by the complexity of simple code elements like classes and methods. Other, more specialized elements are often used as an addition, like annotations (called attributes in .Net), config strings or XML configuration files. Further, often understanding a single API element on its own is not enough because it has to be used in combination with other elements, often following a certain pattern that the programmer needs to understand, like a factory or a fluent interface [71]. Such special elements and patterns have hardly been researched from a usability

Table 2.5: Details on how existing API usability studies were carried out

Title	Users	Tasks	Duration p.p.	Languages	User selection	Environment	Other details
The Factory Pattern in API Design - A Usability Evaluation [58]	12	5	1h+	Java (with Eclipse)	Used on-campus posters and electronic message boards to get participants. Pre-screening survey to ensure at least one year of Java experience. Users were professional developers and software engineers, electrical and computer engineers, and non-technical hobbyist programmers, as well as computer science students.	A screen capturing software was used. Users were thinking aloud.	Presented the tested pattern in as many problem domains as possible to avoid influence of knowledge in a certain problem domain.
Usability Implications of Requiring Parameters in Object's Constructors [182]	30	6	2h15m	C++, C# and .Net	Developers from a Microsoft usability lab, chosen out of 3 different persona (systematic, pragmatic, opportunistic)	Usability lab, experimenter behind one way mirror, test users could speak to experimenter over microphone. Work was captured with a screen capturing software. Test users were thinking aloud.	The cognitive dimensions framework was used for evaluation.
The Implications of Method Placement on API Learnability [184]	10	3	~1h	Java (with Eclipse)	Used on-campus posters and electronic message boards to get participants. Pre-screening of participants via an online survey to ensure sufficient programming experience and knowledge of Java. Users were PhDs, masters and undergraduate students.	A screen capturing software was used. Users were thinking aloud.	Before 2 of the tasks, participants first had to write pseudo code to show their expectations for the task. JavaDoc for the API was presented over a browser window.
An Empirical Study of API Usability [150]	25	5	~70min	Eiffel	10 computer science students, 7 PhD students and 2 post-docs, as well as 6 professional programmers working for various software companies. All had at least one year of experience with object-oriented programming.	Users were thinking aloud. A post test questionnaire was used with 12 questions related to the cognitive dimensions.	
Methods Towards API Usability: A Structural Analysis of Usab. Problem Categories [85]	8	?	1 day	Java (with Eclipse)	Developers with between 1 and 9 years of experience with Java, 7 out of 8 with a university degree, one student.	Users were participating in a whole-day workshop, where they first were shown how to use the API, and then had to solve tasks. Users were captured on video. After the test users were interviewed in 30min sessions.	

point of view, and are not taken into account by any existing code metrics (see section 2.3). E.g. an attribute placed on a class, or the existence of an XML file, would be completely ignored when calculating the complexity of code.

The only pattern that has been evaluated in more detail concerning its usability is the factory pattern [58], as already mentioned above. [35] and [36] identify usability problems with using attributes in .Net, like hidden dependencies between attributes. But unfortunately they don't compare the attribute-based APIs to ones that are not using attributes, so it is unknown whether (or in which situations) an attribute-based approach would be easier to use that something else. In the context of XML there are a few papers dealing with usability [81, 161], but they are either too problem specific or not applicable in the context of programming. To our knowledge there exist no other papers dealing with the usability of special API elements and patterns.

Evaluation of APIs for Specific Problem Domains

There are a few publications where the usability of APIs for specific problem domains is evaluated. [11] presents a usability study about authorization APIs for web applications, and gives recommendations for the design of such APIs. The paper is interesting, though very specific to the topic. [144] analyzes a large number of projects that facilitate the new parallel functions in the .Net Framework, namely the Task Parallel Library (TPL) and the Parallel Language Integrated Query (PLINQ), to find out how developers use these libraries. They present interesting findings about the usability of these libraries, though they are also very specific, and hardly generalizable. [12] presents usability challenges related to enterprise SOA APIs (with a special focus on SAP's web services), but it doesn't give concrete solutions for these challenges.

Code Completion

In section 2.1.2 we identified the IDE, and especially its code completion functions, as an important factor influencing the usability of APIs. To understand how code completion works and how usability is influenced by it, we investigated popular IDEs for Java and .Net. For both there are multiple IDEs available, from which we selected the most popular ones[2] for comparison [169]:

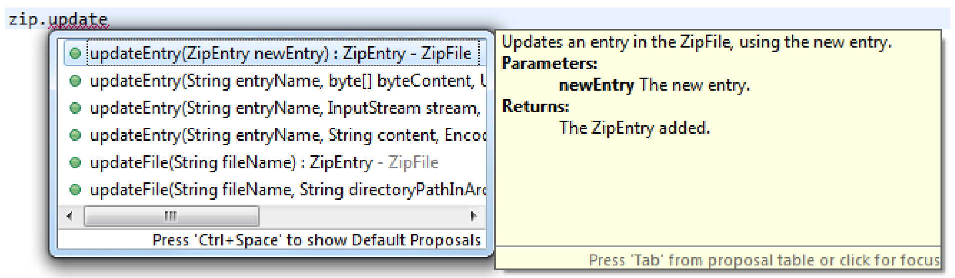
The most popular IDE for Java is *Eclipse*[3]. Figure 2.4 shows a screenshot of the Eclipse code completion feature. For every method of a class, all parameters (including parameter name and type) and the return type are shown, as well as the name of the class where the method is defined. On the right hand side of the method list the documentation for the currently selected method is shown. *NetBeans*[4] closely follows Eclipse in popularity. The code completion window looks the same as in Eclipse, with the exception that the class names are not displayed and the formatting looks a bit different. Another Java IDE is called *IntelliJ*[5], where again there are some small differences like formatting, but despite of that the code completion concerning external APIs works the same as in Eclipse.

[2]e.g. http://java.dzone.com/polls/what-ide-do-you-use-everyday

[3]https://www.eclipse.org/

[4]https://netbeans.org/

[5]http://www.jetbrains.com/idea/



```
zip.update
```

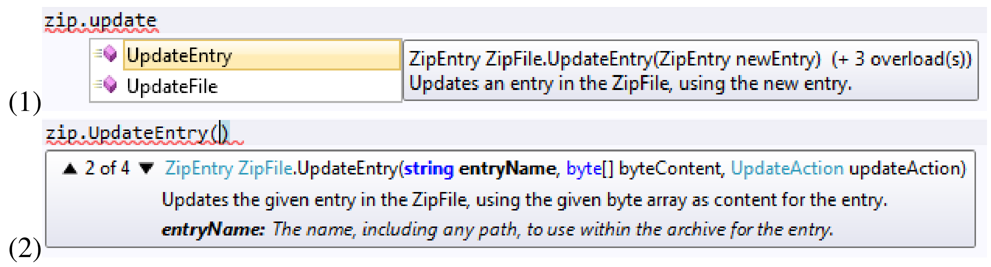
- updateEntry(ZipEntry newEntry) : ZipEntry - ZipFile
- updateEntry(String entryName, byte[] byteContent, U
- updateEntry(String entryName, InputStream stream,
- updateEntry(String entryName, String content, Encod
- updateFile(String fileName) : ZipEntry - ZipFile
- updateFile(String fileName, String directoryPathInAr

Press 'Ctrl+Space' to show Default Proposals

Updates an entry in the ZipFile, using the new entry.
Parameters:
newEntry The new entry.
Returns:
The ZipEntry added.

Press 'Tab' from proposal table or click for focus

Figure 2.4: Code completion in Eclipse: Methods and overloads in a single window.



(1)

```
zip.update
```

- UpdateEntry
- UpdateFile

ZipEntry ZipFile.UpdateEntry(ZipEntry newEntry) (+ 3 overload(s))
Updates an entry in the ZipFile, using the new entry.

(2)

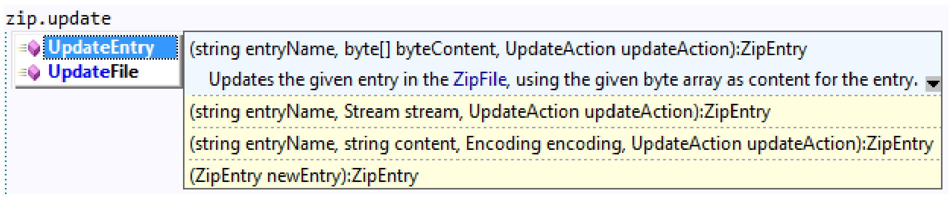
```
zip.UpdateEntry()
```

▲ 2 of 4 ▼ ZipEntry ZipFile.UpdateEntry(**string** **entryName**, byte[] byteContent, UpdateAction updateAction)
Updates the given entry in the ZipFile, using the given byte array as content for the entry.
entryName: *The name, including any path, to use within the archive for the entry.*

Figure 2.5: Code completion in Visual Studio: (1) selection of method, (2) selection of overload.

The most popular IDE for .Net is *Visual Studio*[6]. When compared to the Java IDEs, the code completion shows one big difference: When searching for a method, the list only shows the method names, but not the parameters and return values. While selecting the method, only the parameters of the first overload are shown in the documentation window, but information about the other overloads remains hidden (see Figure 2.5(1)). Only after selecting the method, the overloads can be explored one at a time using the up and down arrows (see Figure 2.5(2)). So, instead of selecting the desired method and overload in one step, Visual Studio splits this procedure into two consecutive steps. This changes when installing *ReSharper*[7], which is a popular addon for Visual Studio that enriches the IDE with a number of helpful coding features, and also enhances Visual Studio's code completion feature. Like in Visual Studio, the code completion window first only shows a list of method names without any information about parameters or overloads. But now, at the same time an additional list of all available overloads is shown for the currently selected method (see Figure 2.6). There are two other .Net IDEs that are well known, though by far not as popular as Visual Studio: One is *SharpDevelop*[8], which is available for free. Despite of a different formatting, the code completion mechanisms are the same as in Visual Studio (whithout ReSharper). The second is *MonoDevelop*[9], which is a multi platform IDE for .Net and its open source variant Mono. Again, the code completion is presented in the same way as in Visual Studio.

[6]http://msdn.microsoft.com/vstudio
[7]http://www.jetbrains.com/resharper/
[8]http://www.icsharpcode.net/opensource/sd/
[9]http://monodevelop.com/



```
zip.update
UpdateEntry    (string entryName, byte[] byteContent, UpdateAction updateAction):ZipEntry
                 Updates the given entry in the ZipFile, using the given byte array as content for the entry.
UpdateFile     (string entryName, Stream stream, UpdateAction updateAction):ZipEntry
               (string entryName, string content, Encoding encoding, UpdateAction updateAction):ZipEntry
               (ZipEntry newEntry):ZipEntry
```

Figure 2.6: Code completion in Visual Studio with ReSharper: Methods and overloads in two different windows.

What all of the IDEs have in common is that the items in the code completion window are generally sorted in an alphabetical order, and method overloads are sorted first by the number of parameters and then alphabetically (starting with the type of the first parameter). Also, most IDEs learn from the API usage of the developer to a certain degree, so that when the developer enters a certain prefix, the selection in the code completion window will automatically jump to the method that was either most often or most recently used with this prefix. Eclipse and IntelliJ additionally change the ordering of elements in the list, so that after a certain number of usages the most frequently used elements will stand on top.

What we see as the most significant difference is the way how methods and overloads are presented. The explored IDEs show three different solutions: The first one (used by all Java IDEs) is presenting all methods and overloads in a single window, including information about parameters and return types (as shown in Figure 2.4). The second one (used by all .Net IDEs except the ReSharper addon) is presenting methods and overloads in two consecutive steps, first only showing the method names, and only after choosing a certain method showing the overloads one at a time (as shown in Figure 2.5). The third one is offered by ReSharper and is a mix between the other two, presenting only the method names in the first window, but also presenting a second window with the overloads for the currently selected method (see Figure 2.6) in form of a list.

In addition to the features included in existing IDEs, there is a number of papers dealing with code completion mechanisms and how to improve them, mainly concentrating on the filtering and ordering of items that are presented in the code completion window. We reviewed such papers in [169]. While several useful and interesting functions have been introduced, not all are relevant for the use of APIs.

Several papers present new ideas for ordering elements in the code completion window: [156] shows how program history can improve code completion. It presents a comparison of 8 different code completion variants, and shows that programmers tend to call methods to which they recently made changes more often, so that presenting these methods at the top of the list can improve programming performance. While this may have an effect concerning the programmer's own code, it doesn't influence the use of external APIs, since a programmer doesn't make changes to them. Another interesting ordering mechanism is presented in [26], suggesting that code completion could learn from examples which API classes and methods are the most important, i.e. an "example based code completion system". In a study it is shown that this system significantly increases the performance of using code completion. Similar results are shown in [139] with an analysis how a code completion window can present the most often used ways of instantiating a class by learning from usage examples and crowd sourcing. A different approach is presented by [93], which suggests to take into account the identifier names that programmers use to guess which methods they want to call most likely, e.g. if the programmer defines a variable named “angle”, he/she might want to call methods that deal with the calculation of angles. [151] and [99] present further improvements by ordering code completion elements, e.g. by taking into account if a method has been defined in a base- or sub-class, and filtering out certain methods that are unlikely to be used in a certain context.

Further, there are several papers that don’t deal with filtering and ordering: [90] introduces a system for code completion from abbreviated input (“abbreviation completion”), which significantly reduces the number of keystrokes needed to find code elements, and to write code in general. For example, for the method call `chooser.showOpenDialog(null)` the programmer would only need to type `ch.opn(n)` and then use code completion. Since the abbreviations are not predefined but calculated on the fly, the method is not restricted to only specifically prepared APIs, but can be used for all APIs. Another interesting code completion mechanism is a querying system presented in [129], where the users can specify input and output parameters and get presented a list of possible methods (or method chains) to reach the specified goals, making it especially easier if the programmer doesn’t know where to look for a certain method or if multiple methods are needed. [147] shows a way to improve code completion by integrating specialized interfaces, e.g. presenting a color chooser when the programmer wants to instantiate an object of type *Color*.

Since our thesis doesn’t deal with improving API usability by implementing new code completion mechanisms, but with evaluating and improving the API itself, it would be interesting be able to make improvements that particularly have in mind how the API is presented in the code completion window. This is especially important because API developers normally don’t have the possibility to change the code completion mechanism, but can of course change their API. Since code completion is directly integrated into the IDE, it is in general rather difficult to change from the outside. A goal of this thesis is therefore to especially take into account how code completion works when defining the factors that influence API usability.

2.2.3 Documentation Usability

Documentation is an important part of every software product, and APIs are no exception. While the first step is of course always to create a software product with good usability (an easy to use product is most certainly also easier to document than a complex one), usability evaluation should not stop with only the product itself, but should in a subsequent step also target the documentation. [76] identifies the quality of the documentation as an important factor for user satisfaction. This is of course also true for APIs: [50] shows that either using the API documentation or searching for usage examples on the web are very common learning strategies for APIs. [13] defines the quality of the documentation as an important aspect of API usability. Even if an API has good usability, with a bad documentation users may still be unable to use it.

Several papers deal with improving the usability of API documentation, most of them suggesting new ways of presenting the documentation to the user (e.g. new navigation tools):

- Apatite [56, 57] is a tool for Java that visualizes associations of API classes and provides a hierarchy-independent search function that approximates the relevance of API elements by the strength of their association to multiple search terms.

- Jadeite [181] is a replacement for Javadoc, which displays most often used classes more prominently, lets users add expected elements as placeholders that point to the actual API elements that should be used, and analyzes examples to find the most common ways how a class is instantiated.
- API Explorer [51] is an addon for Eclipse that leverages the structural relationships between API elements to make API methods or types that are not accessible from a given API type more discoverable.
- eMoose [47] extracts important usage directives from the Javadoc and highlights them while using an API element within Eclipse, making information more visible that otherwise may have been lost in the running text.
- [138] presents a similar approach, by scanning the documentation text for requirements that are not visible directly in the programming language (e.g. "the parameter must not be null" or "the class is not threadsafe") and extracting usage directives from them. These directives can later be used for the implementation of a help tool (no implementation is available yet).
- [31] suggests ways for classifying and recommending knowledge in API reference documentation which should be presented to the user while programming, and shows their use for Javadoc.
- [160] suggests mechanisms for presenting documentation that is relevant to the API elements the user has used in the code already, and analyzes whether existing interdependencies have been fulfilled (no implementation is available yet).

The related work analysis shows that most enhancements are either only available for a single programming language and type of documentation (Javadoc), or have not been implemented at all. They do not cover other kinds of documentation like tutorials and web content, which according to [50] are of equal importance. There are only few papers that present results in this direction: [143] presents a needs assessment for SDK documentation, which can mostly also be applied to API documentation, though the presented results are unfortunately too general and not very surprising, and the assessment has only been done with a single specialized SDK. [52] analyzes which questions developers ask most often when using unfamiliar APIs (e.g. "How do I create an object of a given type without a public constructor?"), and therefore provides important details about what information should be contained in the documentation. But since the goal of the paper is providing better tool support rather than documentation, it does not go into detail about an improved general documentation structure. A more comprehensive work in this direction is [158], which presents a study of API learning obstacles and comes to the conclusion that a majority of obstacles are related to problems with the documentation. It gives several suggestions to improve API documentation, for example: Providing small usage examples for typical API usage patterns involving more than one method is more useful than single-call examples; and a more linear presentation of documentation is often better than fragmented collections of hyper-linked articles. [114] presents additional interesting results, and comes to the conclusion that conceptual knowledge is very important for understanding API documentation. E.g. for an API for Bluetooth communication, the Bluetooth-related knowledge that is necessary for understanding the API should also be included in the documentation.

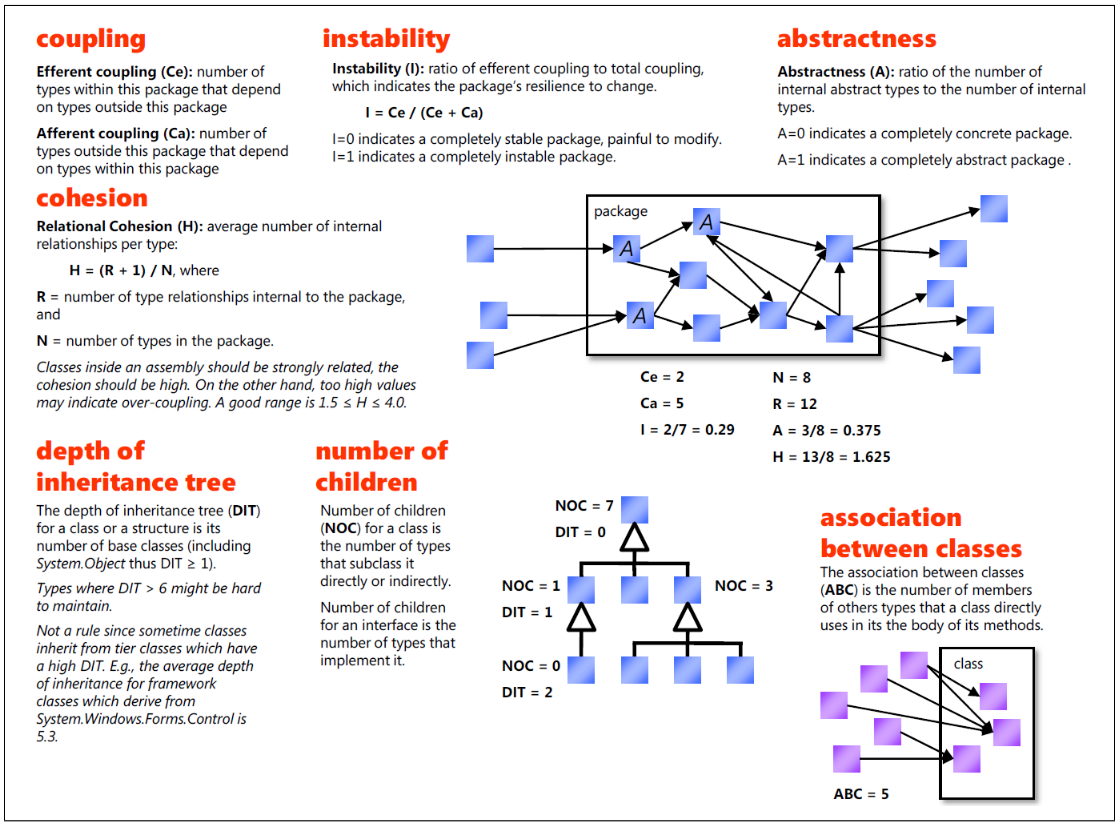
Concerning the evaluation of the usability of documentation, most user-centered tests can be applied without problems. For heuristic evaluation, a special set of heuristics for evaluating documentation has been introduced in [108] (though not for API documentation specifically). There are no reports about a successful automated measurement of usability for documentation. [13] presents several ideas in this direction, and identifies that e.g. the ratio of words per method in the documentation together with the ratio of parameters per method has an influence on learnability. Though the general applicability of these results is questionable, since it is unlikely that a more verbose documentation in always better than a short one. E.g. a verbose but less structured documentation approach is likely to be harder to understand than a compact and highly structured one (e.g. information presented as running text vs. in a structured table). The many possibilities of structuring documentation, as well as the different formats like PDF and HTML make it hard to find suitable usability measures.

In our usability studies we want to investigate the effect of API documentation in contrast to learning an API purely by exploration, since it is obviously a common way of learning and therefore should not be disregarded. For our measurement method we will nevertheless concentrate on the API's usability, since it is the more important aspect to help API developers improve their API designs.

2.3 Automated Software Complexity Measures

Software complexity measures define ways of calculating some aspect of a software's (code's) complexity. Such measures are often used by tools that analyze code quality, e.g. NDepend [179], a popular software measurement tool for .Net to help developers evaluate their code. Figure 2.7 shows an example of some of the measures that NDepend supports, taken from [41]. These measures can for example be used as indicators for the maturity of the software (is it ready to be deployed, or is it likely to still be error-prone? how many bugs are likely to be in the software?), or to track the progress of the software development (to which degree is the product already finished?). It is very important to use metrics with care and to interpret their values thoroughly, as it can be very easy to use them for estimating things that they actually cannot estimate. An example for such misuse is the estimation of developer productivity, which some companies try to do, but simply cannot be done as easily as that (even lines of code has been used for it, which clearly is not a good estimation since longer code is not necessarily better, rather the contrary) [69, 131].

For this thesis we are interested in finding out whether any software complexity measures are able to measure usability. [101] defines complexity as "the degree to which a system or component has a design or implementation that is difficult to understand and verify". Since understandability is clearly an aspect of usability, complexity is therefore also related to / depending on usability. [173] analyzes problems of existing complexity measures and finds many of them in usability-related aspects: A piece of code may be perceived with highly varying



coupling

Efferent coupling (Ce): number of types within this package that depend on types outside this package

Afferent coupling (Ca): number of types outside this package that depend on types within this package

instability

Instability (I): ratio of efferent coupling to total coupling, which indicates the package's resilience to change.

I = Ce / (Ce + Ca)

I=0 indicates a completely stable package, painful to modify.
I=1 indicates a completely instable package.

abstractness

Abstractness (A): ratio of the number of internal abstract types to the number of internal types.

A=0 indicates a completely concrete package.

A=1 indicates a completely abstract package .

cohesion

Relational Cohesion (H): average number of internal relationships per type:

H = (R + 1) / N, where

R = number of type relationships internal to the package, and

N = number of types in the package.

Classes inside an assembly should be strongly related, the cohesion should be high. On the other hand, too high values may indicate over-coupling. A good range is 1.5 ≤ H ≤ 4.0.

depth of inheritance tree

The depth of inheritance tree (**DIT**) for a class or a structure is its number of base classes (including *System.Object* thus DIT ≥ 1).

Types where DIT > 6 might be hard to maintain.

Not a rule since sometime classes inherit from tier classes which have a high DIT. E.g., the average depth of inheritance for framework classes which derive from System.Windows.Forms.Control is 5.3.

number of children

Number of children (**NOC**) for a class is the number of types that subclass it directly or indirectly.

Number of children for an interface is the number of types that implement it.

association between classes

The association between classes (**ABC**) is the number of members of others types that a class directly uses in its the body of its methods.

Figure 2.7: Example of software complexity measures integrated in NDepend, taken from [41]

complexity depending on the programmer's knowledge and experience. Also, while code may be easy to write with a certain structure, it may later be easier to maintain if it were structured differently. For error prediction metrics, [62] finds that whether existing defects really lead to errors strongly depends on how the software is used, and therefore on the behavior of the users. So some defects may never even lead to errors. [204] comes to similar conclusions, saying that the problem of relating defects to failures and therefore to reliability is still unsolved.

So why is usability a term that is never found in literature concerning software complexity measures? It may be because code that a programmer writes is not "used" in the same way an API or GUI is. It actually goes from being written/created to being maintained/improved/extended. So complexity is the more suitable term in this case, though aspects of usability are clearly involved. One could say that usability and complexity are like two different sides of a coin: while both have similar meaning, the former is related to viewing a product from an external viewpoint, while the latter is related to the product's internals.

A categorization of work related to automated software complexity measures is shown in Figure 2.8. The different categories are described in more detail in the following sections. It should be mentioned that in addition to these categories, there is a category of complexity measures that are applied to the design instead of the code. While these measures may not be as

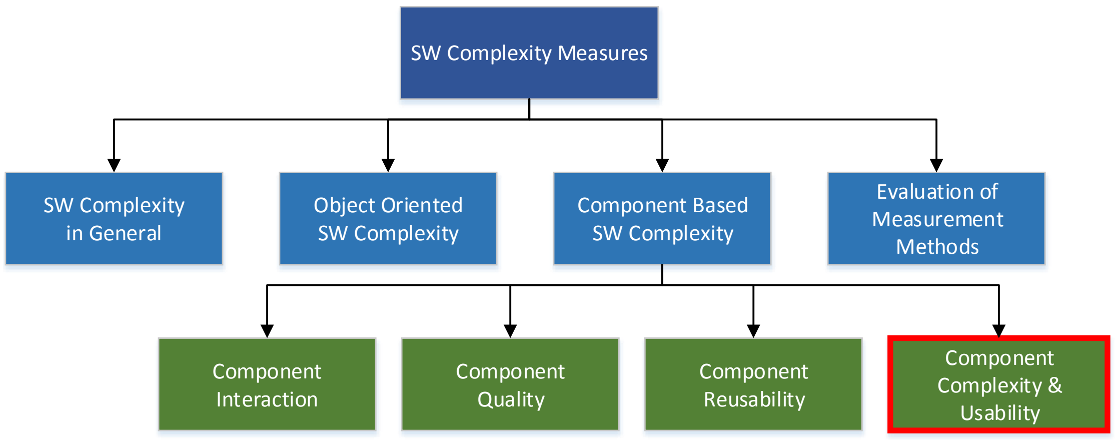

SW Complexity Measures
SW Complexity in General
Object Oriented SW Complexity
Component Based SW Complexity
Evaluation of Measurement Methods
Component Interaction
Component Quality
Component Reusability
Component Complexity & Usability

Figure 2.8: Categorization of Related Work for Software Complexity Measures

accurate as ones that are applied directly to the code, they allow predictions even before the code is written, like the implementation effort, or identification of possibly error-prone modules. The most popular measure in this area is the *function point analysis* [2], which rates different design characteristics of software modules with so called function points. It allows for example to predict the software size and number of errors, to identify modules with a possibly error-prone design, and to track the development progress (function points per work month). Software design measures will not be handled further here, because they do not deal with code and are therefore out of the scope of this thesis.

2.3.1 General Complexity Measures

The most general measures are such that can be applied to any piece of code. Most of them come from an era of programming where functional programming languages were most prominent, which had functions as the only concept for structuring the code (no classes, interfaces, components, ...). The probably most well known of all software complexity measures is *lines of code*, but it has often been criticized to be inaccurate or inefficient [1, 63], starting with the problem of how to actually count the lines in a way that is independent of the formatting. An also very popular and more efficient measure is McCabe's *cyclomatic complexity* [132]. It is based on the idea that a function is more complex, the more different paths can be taken through it. So, the complexity increases for example with every if/else statement, case switch and thrown exception.

While many general complexity measures are already 30+ years old, in recent years some measures have been introduced that are explicitly based on cognitive theory, and therefore better take into account how the complexity is perceived by the programmer. Most of them are based on the idea that the difficulty in understanding software is simular to the difficulty in understanding information, and since software is a mathematical entity that represents computational information, the amount of information contained in software is a function of identifiers that hold the information and operators that perform operations on the information [6]. An example is the *cognitive functional size* [190], which assigns different cognitive weight values to control structures in the code.

The following list gives an overview and short explanation of general complexity measures. When looking at what these measures evaluate, it becomes obvious that they cannot be used to measure API usability, since they only evaluate aspects of a component's internal complexity. But for the user of a component it is not relevant e.g. how many lines of code the component contains, since the user only sees the component's API.

- **Lines of Code (LOC)**: The number of code lines. Different definitions are available, e.g. physical lines of code (exactly as written) or logical lines of code (logical statements), with the latter being preferable because it is independent of the code's formatting.
- **Cyclomatic Complexity** [132]: Complexity of the control flow through a piece of code (how many paths can be taken through the code?). The complexity is equal to the number of decision keywords (e.g. if/else) +1, which for a single method should not exceed a value of 7 (following the cognitive theory that a person can keep track of about 7 different items at the same time).
- **Program Length (N)**[10]: Sum of the total number of operators and operands.
- **Vocabulary Size (n)**[10]: Sum of the number of unique operators and operands.
- **Program Volume (V)**[10]: Describes the size of the implementation, $V = N * log2(n)$. A volume greater than 1000 tells that the function probably does too many things.
- **Difficulty Level (D)**[10]: The difficulty level or error proneness of the program is proportional to the number of unique operators.
- **Program Level (L)**[10]: Inverse to the difficulty level – a low level program is more prone to errors than a high level program.
- **Effort to Implement (E)**[10]: The effort to implement or understand a program, $E = V * D$
- **Time to Implement (T)**[10]: The time to implement or understand a program. [89] defines $T = E/18$ as a good estimation for the time in seconds.
- **Number of Delivered Bugs (B)**[10]: An estimate for the number of errors in the implementation, correlates with the effort to implement the software (E). This can be used as a goal how many errors at least should be found during testing.
- **Fan-In / Fan-Out** [95, 113]: Fan-In is the number of flows into a procedure plus the number of data structures from which it retrieves information. Fan-Out is the number of flows from a procedure plus the number of data structures which it updates. Information flow complexity $IFC = (fan\text{-}in * fan\text{-}out)^2$
- **Number of Faults in Code** [72, 127]: Several metrics have been introduced to predict the number of faults based on the number of lines of code. While [127] suggests the number of faults to depend on the used programming language, [72] comes to a formula which is independent of the used language: The number of bugs per module

[10]part of Halstead's Metrics [89]

$B = 4.2 + 0.0015(LOC)^{4/3}$. To find an optimal balance between number of modules and size per module, this formula suggests an optimal module size of 877 lines of code (though this value is based on functional programming languages, and is most likely different for object-oriented ones).

- **Maintainability Index** [146, 193]: A combined value of other measures. Different variants have been introduced, with the original definition combining Halstead's average effort to implement, cyclomatic complexity, and lines of code per module.
- **Cognitive Functional Size (CFS)** [190]: A measure based on cognitive theory. Gives a rating based on (1) basic control structures (BSCs) in the code by assigning cognitive weights, and (2) the number of inputs and outputs.
- **Cognitive Information Complexity Measure (CICM)** [118]: An enhancement of CFS, additionally rates identifiers and operators. The measure is based on the idea that identifiers and operators become more difficult to understand the deeper down in the code they are (depending on LOC).
- **Cognitive Program Complexity Measure (CPCM)** [135]: Another enhancement/variation of CFS, based on the argument that the number of occurrences of inputs/outputs in the program directly affects the internal architecture and is therefore a better indicator of complexity than only the number of different inputs/outputs.
- **Structured Cognitive Information Measure (SCIM)** [6]: Yet another enhancement of CFS, where a complexity value for each variable in the code is calculated, also taking into account interdependencies between variables.
- **Code/Data Spatial Complexity** [29]: Defines a measure based on the amount of space between definition and usage of a function or variable (measured in LOC). The idea is that the more space between definition and usage, the more time has passed for the user since looking at the definition of the function/variable, so it is more complex to understand and use.

2.3.2 Object-Oriented Complexity Measures

In the area of object-oriented programming languages many metrics have been introduced to evaluate object-oriented aspects of code structure (e.g. class hierarchy). Popular in this area are the metrics of Chidamber & Kemerer [32], which evaluate e.g. the *Depth of Inheritance Tree* and *Number of Children*.

Similar to the general complexity measures, these measures mostly evaluate aspects of a component's internal complexity. For the user of a component it is not relevant e.g. how complex the component's overall class structure is, since the user only sees the component's API, which normally is a very small subset of the classes and interfaces contained in the component. Since a component's API also uses object-oriented concepts like classes and methods, existing measures may still be useful though when adopted to be applied only to the API.

Concerning usability, [53, 54] review existing metrics (especially the metrics suite of Chidamber & Kemerer) concerning their use for assessing the usability of a software system. They come to the conclusion that low values at these metrics imply good usability of a software system. There is, however, no proof given to these conclusions, and other experimental evaluations [13] show that similar metrics cannot be used to rate usability with statistical significance.

The following list gives an overview of object-oriented complexity measures. For each measure it gives a short description and investigates the possible use for evaluating APIs (the latter being marked with $\rightarrow$):

- **Weighed Methods Per Class (WMC)**[11]: The sum of complexities of methods in a class. The original paper leaves the definition open how the complexity of the methods is calculated, therefore many implementations simply use the number of methods as the value for WMC, i.e. they assign a complexity value of 1 for each method (alternatives are e.g. using cyclomatic complexity as in [145]). Classes with many methods are likely to be application-specific and therefore have limited possibility of reuse. $\rightarrow$ Methods also have an impact on the children of a class, which will inherit all base class methods. The number of methods is likely to have an influence on API usability and will therefore need to be investigated.
- **Depth of Inheritance Tree (DIT)**[11]: The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior. $\rightarrow$ This measure is irrelevant for usability, since it doesn't matter for an API user how deep in the hierarchy an API class is placed.
- **Number of Children (NOC)**[11]: The number of direct subclasses of a class. Since inheritance is a form of reuse, a high number of children can point to high reusability, though it could also point to misuse of subclassing. $\rightarrow$ Like DIT, it is not relevant for APIs.
- **Coupling Between Object Classes (CBO)**[11]: The number of other classes to which a class is coupled, either by using the class or by being used by it. Excessive coupling is detrimental to modular design and prevents reuse. Also, maintenance is more difficult because highly coupled classes are more sensitive to changes in other parts of the software. Further, such classes are more difficult to test because a complex test setup is required. $\rightarrow$ For API users, loosely coupled classes may be easier to understand and learn, since it may be easier to view the classes in isolation of each other – so this measure may be of interest for APIs.
- **Response for a Class (RFC)**[11]: All methods that can potentially be executed when a message is received by the class, calculated by the number of methods in the class, and the number of methods of other classes that are called by any method in the class. A large RFC means that testing and debugging the class can be complicated since understanding the class requires a lot of effort. $\rightarrow$ The RFC of an API class is irrelevant since users are not required to debug API classes or understand their internals.
- **Lack of Cohesion in Methods (LCOM)**[11]: Measures the cohesiveness of methods within a class, which is desirable since it promotes encapsulation. LCOM is calculated by the number of non-similar method-pairs minus the number of similar method-pairs. Two

[11]part of Chidamber & Kemerer's metrics suite for object-oriented design [32]

methods are similar if there are one or more instance variables that are used in both of them. A lack of cohesion implies that a class may have multiple responsibilities (violating the single responsibility principle [130]) and should be split into multiple classes. $\rightarrow$ Since classes following the single responsibility principle are also potentially easier to understand, this measure may be of interest for APIs.

- **Average Method Complexity (AMC)** [145]: This metric calculates the cyclomatic complexity per method to create an average method complexity value per class. Although quite simple, the paper shows that it allows an accurate prediction of a class' error-proneness, and provides better results than measures like WMC or LOC. $\rightarrow$ Since the internal complexity of methods is rated, there is no relevance for usability.
- **Metrics Suite for Class Complexity** [134]: This metrics suite tries to especially improve the WMC metric by providing additional and enhanced metrics for class complexity, including mean method complexity (MMC), standard deviation method complexity (SDMC) and number of trivial methods (NTM). [145] shows that SDMC, like AMC, is a good predictor of a class' error-proneness. $\rightarrow$ Like for the previously described similar measures, there is either no or only very minor relevance for usability.
- **Metrics for OO Design (MOOD)** [24]: Defines a set of 6 different metrics, being method-/attribute hiding factor (MHF/AHF), method/attribute inheritance factor (MIF/AIF), polymorphism factor (POF) and coupling factor (COF). $\rightarrow$ As the names suggest, these metrics highly rely on internal information that is not relevant for a component's API. Their main goal is rating defect density and rework (effort for correcting defects).
- **Lorenz and Kidd's OO Metrics** [128]: A set of class- and operation-oriented metrics, including the class size (number of methods and attributes), the number of operations overridden by subclasses, the "specialization index" (degree of specialization of a subclass), average size of an operation, and average number of parameters for an operation. $\rightarrow$ Of these metrics, only the class size is likely to have an influence on API usability (similar to WMC), all other metrics deal with internals not related to APIs.
- **Quality Model for Object-Oriented Design (QMOOD)** [9]: Defines a set of 11 metrics and maps them to the quality attributes reusability, flexibility, understandability, functionality, extendibility and effectiveness. Understandability is defined as being negatively influenced by abstraction, coupling, polymorphism, complexity and design size, and positively influenced by encapsulation and cohesion. $\rightarrow$ The concrete metrics behind these attributes are similar to the ones described above, and rely strongly on the internal class structure, so they are not directly relevant for APIs.
- **OO Cognitive Weight Metric** [136]: This metric maps the cognitive functional size measure [190] to classes by simply summing up the cognitive size values of the methods. $\rightarrow$ Cognitive size is an internal measure and therefore has no relevance for usability.
- **OO Spatial Complexity** [30]: Maps the spatial complexity measure [29] to object-oriented code, which is based on the idea that method calls are more complex the more space (measured in LOC) is between declaration and usage. Though with modern IDEs in mind, where any user can just use commands like "go to declaration" or "find usages", and

switch around between classes, this idea sounds rather unrealistic. $\rightarrow$ We therefore don't see a relevance for usability. Unfortunately, the paper doesn't contain any studies to validate the measurement results.

- **Maintenance Prediction Metrics** [8]: A set of three metrics for predicting maintenance effort, which are interaction level (IL), interface size (IS) and operation argument complexity (OAC). The metrics are based on rating parameters by their complexity (e.g. Boolean=0, Integer=1, Float=2, ...). $\rightarrow$ Although interfaces are rated, the given parameter rating is unlikely to be fitting for usability.

2.3.3 Component Based Complexity Measures

There is a number of metrics that have been introduced for measuring different aspects of software components. Many of these measures combine and extend previously introduced general and object-oriented measures. They can be split into different categories:

Component Quality

A number of quality models for software components have been introduced [3, 14, 15, 107, 172, 178]. Many of them propose metrics and map them to different quality attributes (also usability), but most metrics are too unspecific and/or cannot be measured automatically. E.g. [3] and [14] both propose time to learn (use, configure, administrate) as a metric for learnability and the presence and readability of the documentation as a metric for understandability, which makes sense, but these values cannot easily be acquired. We will therefore not analyze these metrics further.

Component Interaction

Measures of this category deal with evaluating the overall complexity of a component based system by evaluating the complexity of the components' dependencies and interactions. The measures are often based on graph theory, where the components are the nodes, and dependencies/interactions are directed edges. The interaction between components is not directly relevant for API usability, since it does not matter for an API user whether the API internally consists of multiple components or not.

- **Component Interaction Metrics** [111]: A set of 5 different metrics that measure the complexity of component interaction, like the percentage of component interactions and total interactions performed. Unfortunately no evaluation of the measures is given, and it is unknown which quality aspects they actually aim to measure.
- **Component Dependency and Interaction Density** [79]: Two metrics that evaluate the number of dependencies between components, as well as the number of interactions compared to the total number of components. The metrics claim to evaluate the complexity of a component based system, but again there is no evaluation in that direction, only a formal validation of the method's integrity.

- **Component Coupling Factor (CCOF)** [191]: A component coupling metric to measure the maintainability of a component-based system, also including remote components.
- **Component Density and Criticality Metrics** [120]: A suite of metrics to evaluate density and criticality. Density metrics are e.g. packing density (average component size) or interaction density (number of ingoing/outgoing interactions). Criticality metrics aim to identify critical components, e.g. by their size and number of interactions. The measures are mapped to several quality attributes, including usability, though the rationale for this mapping is unclear and therefore questionable.

Component Complexity

Measures in this category deal with measuring the complexity of a component in general, either from an external (only the component's API is considered) or internal (complete component code is considered) view. In many cases, it unfortunately doesn't become clear what the actual goal of the measures is, as it is not explained what "complexity" means in terms of more concrete software quality attributes like understandability or error-proneness. Also, most of these measures do not provide any evaluation. For these reasons, we don't see a direct relevance of any of them for usability.

- **Structural Component Complexity Metric** [205]: Defines a metric to measure the complexity of a software component's internal structure, based on class interdependencies, method complexity, attribute complexity and the amount of messages passed between classes. No validation or mapping to quality attributes is done.
- **Component Complexity Metric** [176]: A complexity metric evaluating the internal and external complexity of a software component. It is based on the complexity of attributes, methods and interfaces. The method is evaluated with a number of JavaBeans, but unfortunately the validation is insufficient.
- **Interface Complexity Metric** [80]: This metric defines interface complexity as a combination of the complexities of the interface signature, constraints and configuration. Although the idea sounds reasonable, any kind of evaluation is missing, and no concrete approach on how to apply the metric is formulated.
- **Software Component Defect Prediction** [207]: A metric for predicting defective software components, based on a combination of three code complexity metrics: lines of code, cyclomatic complexity [132] and Halstead's program length [89]. Common classification techniques like Bayesian Networks are used to combine the three metrics for optimal defect prediction, reaching a prediction accuracy of up to 74%.

Component Reusability and Customizability

Several measures have been introduced to evaluate reusability and customizability. They aim to help developers to identify components that are suitable for reuse, and to increase the reusability of components. The measures show certain relations to usability, which are investigated in the following list:

- **Component Reusability Metrics** [192]: A suite of 5 metrics for reusability, which are applicable to black-box components. Reusability is mapped to understandability, adaptability and portability. Two of the metrics are related to understandability, which are the existence of meta-information, and the rate of component observability. The metrics are implemented for JavaBeans, where the first metric refers to whether a BeanInfo class is provided, and the second to the rate of fields with getter methods. While the existence of meta information may be a reasonable usability metric, it may be rather inaccurate because the understandability of the actual content of the information is not rated. Also, the implementation of the measure is very specific to JavaBeans. For these reasons, the relevance for usability is limited.
- **Component Quality Metrics** [34]: Among others, this quality metrics suite defines two metrics for reusability and customizability, though an automated measurement doesn't seem possible (e.g. one metric contains the "number of methods for attribute/behaviour/-workflow customization", but there is no information on how to identify the methods that fall into these categories), which makes it irrelevant for us.

Component Usability

To our knowledge only two papers so far have dealt with measuring the usability of software components and their APIs [13, 153].

[13] examines the question if a proposed set of measures that seem to logically relate to usability can be used to measure any sub characteristic of usability with statistically significant relevance. Over 30 base metrics are proposed, covering aspects of the software component as well as its documentation. In five consecutive studies, users were questioned after having used certain software components. From the results three combinations of merics are identified that are related to usability characteristics:

- Understandability relates positively to the ratio of HTML files in the documentation per functional entity, and negatively to the ratio of return values per method.
- Learnability relates positively to the ratio of words in the documentation per method, and negatively to the ratio of arguments per method.
- Operability relates positively to the ratio of words in the documentation per configurable parameter, and negatively to the ratio of return values per method.

In the conducted studies, these measures showed statistically significant similarity to the usability perceived by the users (measured with a questionnaire). While these results are very interesting, and the studies seem to have been conducted thoroughly, relying only on questionnaires weakens these results. As stated in [142], the user's opinions are not always objective, and the results also strongly depend on their understanding of the questions. Further, since the measures rely on the documentation, they are not suited to evaluate usability when the API is still being developed (since the documentation usually has not yet been created at this stage).

A different approach is taken in [153], where a set of 9 metrics is introduced that are based on existing API design guidelines. The metrics are easy to apply and relate very well to the respective guidelines. The authors also give good examples on how to use them in practice. Examples for these metrics are:

- API parameter list complexity index (APXI): Rates the complexity of the parameter lists of the API's methods, rated by the overall number of parameters per method, as well as the number of consecutive parameters with the same type.
- API method name confusion index (AMNCI): Evaluates if there exist too many method with nearly identical names, by splitting the method names into terms, removing default terms like "get" and "set", and then counting how often equal terms occur.
- API exception specificity index (AESI): Evaluates if used exception throwing classes are too general, by rating the distances to root and leafs in the inheritance tree of the exception class.

There are several problems that apply to the metrics in both [13] and [153]. Both rely mostly on evaluating properties of the API's methods. Since an API doesn't only consist of methods, but also other programming concepts (e.g. annotations, see for example the APIs introduced in chapter 5), this poses the problem that the metrics cannot give a comprehensive overview over the usability of an API. What we see as the biggest problem though is, that none of the metrics take the *context of use* into account (all are executed on the API/documentation as a whole), although [13] even clearly emphasizes its importance. While all the metrics are ratios and therefore independent of the API's overall size, some parts of an API may be easier to use than others. E.g. if only the most important parts of the API are extensively documented, all parts of the API would suffer. This not only leads to inaccurate measurement, but also makes it more difficult to identify which areas in an API need to be improved.

Nevertheless the metrics give valuable input that complements our own research.

2.3.4 Comparison of Measurement Methods

To find out if any of the measures can be used to measure usability, Table 2.6 shows a categorization of the measures concerning to which part of the code the measure is applied, and which characteristics are measured. The most often measured characteristics are:

- Size: This is the most basic measurable characteristic, trying to estimate how large a piece of software is or will be, e.g. for comparison with other software, or to estimate the implementation effort.
- Understandability: Many measures aim to evaluate the understandability of code, since it is an important aspect of the widely used general term "complexity".
- Error-proneness: Measures for error-proneness are used for predicting the number of errors, to check how many errors should be found before deploying a software product, to identify the parts of the software that require the most testing effort, and to predict the effort required to correct existing errors ("rework").

- Reusability: Reusability is especially interesting for object-oriented and/or component-based code, and deals with the question how easily a class or software component can be reused.
- Others: There are several other quality characteristics being measured by some metrics, like flexibility, functionality, extensibility and effectiveness. These are only few in number, and not of special interest for this thesis, are are therefore combined in a single category in the table.

The categorization was done by analyzing each published metric for what the authors claim can be measured by it. In most cases such statements were easy to find in the respective publications. Sometimes additional information was available from other publications that analyzed the correlation of the metrics to specific factors – in such a case this information was also used for the categorization. For a few metrics, the authors just stated that the metric measured "complexity", and gave no more specific information. In such a case we tried to give a good guess on what the authors aimed to measure, which most often resulted in a categorization of either understandability or error-proneness. Metrics suites, which consist of several metrics (e.g. [32]), are shown as a single line in the table.

What is especially problematic with some of the metrics, is their either insufficient or completely missing validation. Many papers like [176] use other metrics to show whether their own metric is valid by checking for correlation. This obviously poses the problem that the validity of the metric completely depends on whether the other metric that is used for checking is valid itself. Further, if the two metrics correlate, the question is why there was even a need for a new metric, as it seems that an already existing metric provides the same results. The problem is even bigger when the evaluation is poorly done – e.g. [176] comes to a weak correlation of -0.31 to an existing metric, but despite its weakness uses this as proof that the metric works. A few papers don't even provide any kind of validation, and for example simply state that "the proposed metric appears to be logical and fits the intuitive understanding" [80]. While most older metrics provide sufficient validation, especially in recent years a larger number of metrics have been introduced that are weakly validated. Because of the increasing popularity of component based systems, most of these weak metrics are component based metrics. All such metrics, where validation is either missing or highly unrealistic, are marked in table 2.6 with ? .

Especially interesting for this thesis are the measures that can be applied to a component's API, which are only very few, as the table shows. Three of them are object-oriented measures. They are marked with ~ because they are not originally targeted to APIs especially, and only some of the measures in these suites make sense for APIs. One of them is even used by a tool called Metrix to measure and visualize the usability of APIs [46], but unfortunately no proof is given that the results can really be related to usability. Further, three component-based measures are dealing with the component's API. Two of them unfortunately provide no or insufficient validation, and integrate ideas that do not seem fit to measure usability, so they can at best be used as sources of inspiration. Only two papers stands out [13, 153], providing measures clearly aimed to evaluate usability and sufficiently validated, but suffering from other disadvantages as already discussed in section 2.3.3. What is particularly interesting is that, even among the measures targeted to APIs, there exist absolutely none that try to take the context of use into

Table 2.6: Categorization of Software Complexity Measures

	Applied to...				**Measures...**				
Measure	**Method Internals**	**Component Internals**	**Component Interaction**	**Component API**	**Size**	**Understandability**	**Error-proneness**	**Reusability**	**Other**[1]
Lines of Code	+				+				
Cyclomatic Complexity [132]	+				+	+	+		
Halstead's Metrics [89]	+				+	+	+		
Fan-In / Fan-Out [95, 113]	+				+	+	+		
Number of Faults in Code [72, 127]	+						+		
Maintainability Index [146, 193]	+					+	+		+
Cognitive Functional Size [6, 118, 135, 190]	+				+	++			
Code/Data/OO Spatial Complexity [29, 30]	+	+				?			
Chidamber & Kemerer's OO Metrics [32]		+		~	+	+	+	+	
Average Method Complexity (AMC) [145]		+				+	+		
Metrics Suite for Class Complexity [134]		+				+	+		
Metrics for OO Design (MOOD) [24]		+					+		
Lorenz and Kidd's OO Metrics [128]		+		~	+	+	+		
Qual. Model for OO Design (QMOOD) [9]		+				+	+	+	+
OO Cognitive Weight Metric [136]		+			+	++			
Maintenance Prediction Metrics [8]		+		~	+		+		+
Component Coupling Factor (CCOF) [191]			+			+	+		
Component Interaction Metrics [111]			+			?	?		
C. Dependency and Interaction Density [79]			+			?	?		
Comp. Density and Criticality Metrics [120]			+			+	+		+
Component Reusability Metrics [192]			+			+		+	
Component Defect Prediction [207]			+				+		
Component Quality Metrics [34]			+		+			+	?
Structural Comp. Complexity Metric [205]			+			?	?		
Component Complexity Metric [176]			+	+		?	?		
Interface Complexity Metric [80]				+		?			
Usability of Software Components [13]				+		+			+
Structural Measures of API Usability [153]				+		+			+

1) Other quality attributes, like flexibility, functionality, extensibility, effectiveness

account. We can therefore conclude that there exist no code complexity measures that are able to measure usability with respect to the requirements defined for our measurement method, though some measures may provide valuable input for our method.

2.3.5 Validation of Measurement Methods

Many papers (e.g. [29, 32, 119, 120, 136, 176]) validate the integrity of measures using a set of properties that are very popular in this area, called *Weyuker's properties* [198]. Although the properties have originally been introduced for general code complexity measures like cyclomatic complexity, they have also been used for object-oriented and component-based measures. This has been criticized by some [177], as not all properties seem suitable for object-oriented systems. But what we see as the biggest problem is that some papers are eager to use Weyuker's properties, but do not even deal with the question *what* their measure actually measures, or if what it measures has any real-world meaning, e.g. [30]. [137] shows that in some cases the properties were even misused (or just misunderstood) just for being able to state that the introduced measure satisfies all properties, although this may not in all cases be necessary, since even popular measures like cyclomatic complexity don't satisfy all of them. What should be more important is that the measure provides meaningful results. [28] shows that satisfying Weyuker's properties doesn't necessarily mean that the measure measures anything useful – it is after all just a validation of the measure's integrity. We agree that checking the usefulness of measurement results is more important than satisfying a set of formal properties, and will therefore not use Weyuker's properties, but rather concentrate on evaluating our measurement method by comparing it with the results from usability studies.

Other sets of properties have been introduced, though none of them are as well known and widely used as Weyuker's Properties. E.g. [22] introduces different sets of properties for metrics with different purposes. For example, two properties of size metrics are that the result is always non-negative, and that the sum of the sizes of two programs equals the size of the two programs combined. A comparison of different property sets can be found in [22].

CHAPTER 3

API Usability Measurement Approach

This chapter describes the approach for measuring API usability. First we identify measurable characteristics of API usability. Then we define our measurement approach, which is based on measurable concepts and properties, and is called the *API concepts framework*. To identify properties that potentially influence API usability and can be measured automatically, we conduct a literature review. The identified properties are what will then further be evaluated with usability studies.

The measurement approach was first published in [167], and later in greater detail in [171].

3.1 Measurable Characteristics of API Usability

According to [13], at least three measurable characteristics can be identified as being related to API usability: The *complexity of the problem*, the *complexity of the solution* and the *quality of the documentation*. Since a comparability between APIs is only relevant for solutions for the same kind of problem, it can be assumed that all of them share the same *complexity of the problem*, so there is no particular need for evaluating this characteristic. Furthermore, while documentation has proven to play an important role for usability (see section 2.2.3), especially in earlier development phases it will not be fully available for evaluation. When a developer designs an API, he/she will evaluate it by writing some usage examples against this API, but not already create its documentation. Therefore we will here focus on only one of these three characteristics, which is the *complexity of the solution* built with a given API and for a given scenario.

3.1.1 Measuring the Complexity of the Solution

We propose that the *complexity of the solution* can further be split into the following measurable sub-characteristics:

Interface Complexity

Interface complexity describes the complexity of the elements of an API (classes, methods, fields, ...) that are used for implementing a given scenario. The fewer elements a developer must use, and the less complex they are, the better the usability of the software component will be. What is important is the focus on how an element is *used*: For example, when a method needs to be called, a developer needs to (1) find the correct method, and (2) write down the code to correctly use the method (e.g. fill out the method parameters). Relevant actions include

- basic API actions, like instantiating a class, calling a method and accessing a field,
- more advanced actions like implementing an interface, inheriting a class or placing an annotation,
- and also actions that are not related to a single API element or that are not covered by "normal" API usage, like using a fluent interface.

We believe that this is a main factor of API usability, and that this cannot yet be evaluated by any existing automated software measure.

Implementation Complexity

Implementation complexity describes the complexity of the resulting code when implementing a given usage scenario. This is especially important for comparing APIs that are not equal in functionality. If a needed feature is not supported by a certain API, the system developer will need to implement it himself, leading to much higher implementation effort.

For example: For a remote messaging scenario, functionality for lookup and replication is required. From components A and B, both support lookup, but only B supports replication. Although A may have a lower interface complexity and therefore seem easier to use, the better component for the given scenario is B, because the need to implement replication on your own makes A more expensive in the end, since it makes the resulting code much more complex.

There exist different measures that have proven to be very useful to show the complexity of code, like the ones described in section 2.3. We suggest to use the following ones to evaluate different quality aspects of the code:

- For understandability and complexity: Cyclomatic complexity [132], which measures the complexity of the code's control flow.
- For maintainability: The maintainability index [146, 193], which combines different complexity factors into one metric.
- For object-oriented code quality aspects: Chidamber & Kemerer's object-oriented metrics [32]. While there are multiple object-oriented metrics suites that could come into consideration (e.g. [128, 134]), we simply suggest this one because it is the one most widely known (there exists no clear proof whether any of the suites provides generally better results than the others).
- To estimate the size of the solution: The number of logical lines of code, in other words the number of statements [2].

Setup Complexity

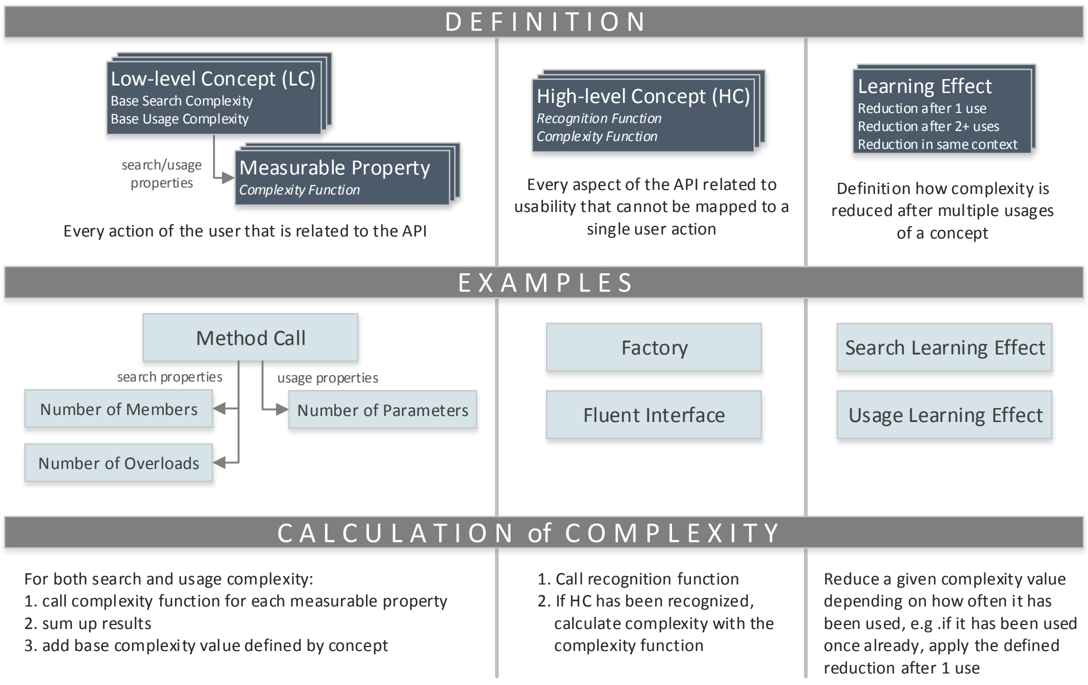
Setup complexity includes any action that needs to be taken so that a given API can be used and the developer is able to create software with it, for example programs that need to be installed and configured. This kind of complexity is supposedly rather difficult to measure (especially automatically), because unlike interface complexity, there exist no predefined concepts – the actions that need to be taken can be very diverse. The following actions need to be taken into account for example:

- Installing software: Certain software components require installation before they can be used (some even require additional installations to fulfill pre-requirements). The more effort a developer has with installation, the more complex it is to get the software component ready for use. This also includes things like setting up registry values or environment variables.
- Adding a reference to a library: When using an API, the developer needs to add a number of additional libraries (assemblies in .Net, jar files in Java). Needing to add no library at all is of course easiest (which is the case when using a feature that is built directly into the programming language, for example WCF in the .Net Framework), whereas adding multiple references may require additional time.
- Custom build/run configuration: Some libraries require using a custom configuration for compiling or running the resulting solution. Developers need to set up this configuration, e.g. by using a special template or by defining certain configuration parameters themselves. This can also lead to difficulties when debugging, if the standard debugger is not able to deal with this special configuration.

Similar to the *quality of the documentation*, setup complexity is mainly important when different APIs are compared, and won't play a big role when a developer wants to evaluate the usability of his own API. It will therefore not be a main research target in our thesis. To our best knowledge there exists no literature dealing with this kind of complexity.

3.1.2 Similar Approaches in Literature

What is most unique about our approach is that (1) it takes the context of use into account (i.e. the task to be solved, as well as the developer's experience), and (2) that both the API and the resulting code are evaluated. Existing API usability metrics [13, 153] only evaluate the API itself, but not the resulting code, and ignore the context of use. There are only very few mentions of similar ideas: [80] defines interface complexity as a combination of the complexities of the interface signature, constraints and configuration, where signature conforms to our definition interface complexity, and configuration to setup complexity. But the details of these aspects have no relation to usability, and the complexity of the resulting code is not taken into account. [46] mentions facitilating API usage examples as an idea for future work for automated usability measurement based on the context of use, but the idea seems to have never been pursued further.



DEFINITION
Low-level Concept (LC)
Base Search Complexity
Base Usage Complexity
search/usage properties
Measurable Property
Complexity Function
Every action of the user that is related to the API
High-level Concept (HC)
Recognition Function
Complexity Function
Every aspect of the API related to usability that cannot be mapped to a single user action
Learning Effect
Reduction after 1 use
Reduction after 2+ uses
Reduction in same context
Definition how complexity is reduced after multiple usages of a concept
EXAMPLES
Method Call
search properties
usage properties
Number of Members
Number of Overloads
Number of Parameters
Factory
Fluent Interface
Search Learning Effect
Usage Learning Effect
CALCULATION of COMPLEXITY
For both search and usage complexity:
1. call complexity function for each measurable property
2. sum up results
3. add base complexity value defined by concept
1. Call recognition function
2. If HC has been recognized, calculate complexity with the complexity function
Reduce a given complexity value depending on how often it has been used, e.g .if it has been used once already, apply the defined reduction after 1 use

Figure 3.1: The API concepts framework: Definition of elements, examples and calculation

3.2 The API Concepts Framework

Our goal in this thesis is to define an extensible framework for measuring *interface complexity*, which we believe to be the main and still unresearched characteristic of API usability. While the other two characteristics are also important, *implementation complexity* can already be measured with existing methods, and *setup complexity* is only important while preparing the API to work with it. Also, interface complexity alone is already sufficient for API developers to evaluate their interfaces, or to compare two APIs that have similar functionality, and require a similar amount of code to be written.

We claim that an API is more complex to use, the more of its *concepts* you need to use, and the more complex these concepts are. We further split concepts into *low-level* and *high-level* concepts. Together with *learning effects*, they build the foundation of our framework. Thus we name it the *API Concepts Framework*. Figure 3.1 shows a definition for each element, as well as concrete examples and how the complexity is calculated. Figure 3.2 illustrates the inputs and outputs of the framework.

3.2.1 Low-level Concepts (LCs)

A low-level concept (LC) represents a single API-related action that the developer needs to take when implementing a given scenario, e.g. instantiating a class or calling a certain method. Such an action basically consists of two steps: (1) searching for the right concept to use (e.g.

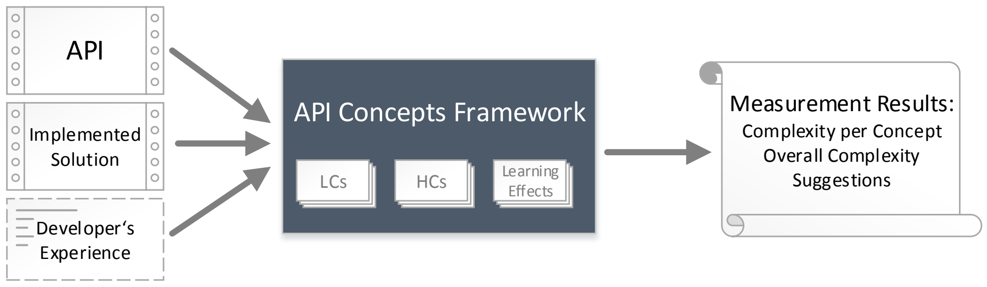


Figure 3.2: The API concepts framework: Inputs and outputs

looking for a suitable method using the code completion window in the IDE) and (2) using the concept (e.g. correctly filling the method parameters). Because of this the LC is split into the two different aspects *search* and *usage* complexity. For each the LC contains a static base complexity value, as well as a list of measurable properties.

Measurable properties represent the dynamic characteristics of a LC that influence usability. For example, the effort for searching a certain method depends on the number of other members in the class, so this is a measurable search property for the concept method call. Each measurable property defines a complexity function, which specifies how the complexity is calculated.

For an API developer, the goal of usability measurement is of course to find areas where an API can be improved. Because of that, a measurable property additionally allows defining suggestions. For example, a suggestion could be to reduce the number of method parameters, depending on a predefined limit. This allows a developer to get concrete clues about possible usability improvements.

3.2.2 High-level Concepts (HCs)

The usability of an API is also influenced by factors that cannot be directly related to a single user action, like design aspects of the API that influence how well a user can understand it. Such aspects are represented by high-level concepts (HC). A good example are design patterns like *factory* or *fluent interface*.

An important issue of HCs is their recognition. For LCs this is not a problem, since every LC can be clearly recognized by a certain code element (e.g. class or method). On the other hand, recognizing the presence of a design pattern is a more difficult task. Therefore a function is required to recognize the presence of the HC. For example, a fluent interface could be recognized by the presence of a fluent method chain (multiple API methods called in a row).

Like a measurable property, a HC further needs a function which calculates the complexity. This function can do two things: (1) calculate a standalone complexity value for the HC, (2) alter the complexity of LCs. An example for the latter is that using a method of a fluent interface for the first time is more difficult than a normal method call, because the developer needs to understand how to place it in a chain of methods. To help API developers finding ways to improve their APIs, a HC also allows to define suggestions.

3.2.3 Learning Effect

An important part of usability is the user's experience. A user who has already learned how to use an API will perform very differently than one who uses it for the first time. So a concept will have a reduced complexity, depending on the number of previous uses (assuming that the uses happen within a reasonable timespan). This is depicted in Figure 3.1 as *learning effect*.

The results from our usability studies (see chapters 4 and 5) show that when a user uses a concept for the third time, he/she has learned how to use it completely, and there is no significant further improvement after that. Therefore, a learning effect defines the complexity reduction after one use, and after two or more uses. Further, there is an even stronger complexity reduction when a concept is used multiple times within the same context (solution), which is mainly because in this case users often just copy/paste the code of the first usage. Therefore a third property of the learning effect is the reduction for consecutive usages within the same context.

For an API user, the learning effect allows to compare an API which he/she has much experience with, to one that he/she has never worked with before ("should I use this new API, or stay with the one I am already familiar with?"). For an API developer, it allows to evaluate how complex the API is both for beginners and for experienced users.

Later we will show that the actions search and usage have a very different learning effect (see chapter 4). With the given definition, it is easily possible to define separate learning effects for both actions.

3.2.4 Actions for Measurement

As shown in figure 3.2, the framework has three different inputs:

- The API to be evaluated, e.g. in the form of a JAR file in Java, or a DLL file in .Net. The API's source code is not necessary – using reflection, the framework can evaluate the API structure even from the compiled code. Also, it is not necessary that the functionality behind the API is already implemented for doing the evaluation.
- The solution that has been implemented for a specific scenario, which represents the context of use. This could e.g. be a usage example, or even the code from a unit test, which is often already present in the beginning, when test driven development is used.
- Optionally, information about the developer's experience, formatted as a list of concepts including the information how often the developer has used them already. If no such information is given, the calculation assumes a developer that has never used the API before.

An API developer can start measuring the usability of his/her API as soon as these inputs are satisfied, i.e. when the basic API structure has been implemented (with no real functionality yet required), and when the code for some usage examples has been written. The framework carries out the following actions for measurement, in the specified order:

1. extract a list of used LCs from the implemented solution / usage example
2. for each LC calculate search and usage complexity with the steps shown in Figure 3.1

3. recognize HCs
4. calculate complexity for each recognized HC, adjust the complexity of LCs
5. apply the appropriate learning effects to the results from steps (2) and (4), according to the developer's number of previous usages, as well as usages within the given solution

The output of the framework is a list of the concepts and their complexity values, as well as suggestions defined by the measurable properties and HCs. The sum of all complexity values gives an overall comparable result.

The defined actions are independent of concrete concepts, making the framework easily extensible with new concepts and measurable properties, with the only requirement that they follow the defined structure.

3.3 Potential Factors Influencing API Usability

In this section we try to identify factors that potentially influence the usability of APIs by reviewing existing literature. This includes API usability studies, API design guidelines and reports about API usability problems. Table 3.1 lists 22 usability-related aspects that we have identified. The left column lists the usability aspects that have been found in literature. For each aspect, the right column lists potential measurable properties, which are numbered and categorized into two different types: $\boxed{\text{N..}}$ properties count something, e.g. the number of members in a class. For these properties, a small number is always better for usability than a large number, e.g. it is more difficult to find a member in a class, the more members the class has. $\boxed{\text{R..}}$ properties describe the relation between different API elements, e.g. a method with parameters of undefined type is more difficult to use than one with parameters of a concrete type. The aspects are ordered by a combination of how important we believe they are, and how well we think the identified measurable properties can provide meaningful results. We later use this ordering to prioritize which properties should be evaluated with usability studies.

The API Concepts Framework shall be applicable to modern object-oriented programming languages and therefore concentrates on Java and C#, which are the two most widely used languages in this area. To reach that goal, the framework shall also take usability-related differences between the two languages into account, and rate the complexity accordingly depending on the language.

Table 3.1: Identification of measurable properties from usability aspects

API Usability Aspect	Potential Measurable Properties (MPs)
An API should be minimal, without imposing undue inconvenience on the caller. Unnecessary exposure of internal API logic, like internal classes and methods, can confuse the user and therefore lead to bad usability. "When in doubt, leave it out." [20,45,85,94,188,211]	N01 the number of overall classes in the API N02 the number of classes in the package/namespace N03 the number of members in a class
The developer should not be required to do anything the module could do itself [20]. For example, if there are multiple API calls which the developer always needs to call in the same sequence, it may be better to provide a single API method for it.	N04 the number of required low-level concepts Note: Additionally, this aspect is measured by *implementation complexity*.
Users should not need to explicitly instantiate more than one type in the most basic scenarios. "The number of customers who will use your API is inversely proportional to the number of `new` statements in your simple scenarios." [45]	R01 the usage of a class is one of the most complex concepts (to motivate minimizing the number of classes)
Classes for different tasks should be placed in different packages/namespaces, and the main package/namespace should only contain classes for common scenarios. This makes the API easier to explore for developers that learn by experimenting. [45,188]	N02 (see above) R02 property N02 has a bigger influence on complexity than N01
The number of method parameters and return values has a strong influence on usability. [13,20,85,211]	N05 the number of method parameters and return values
The naming of API elements is very important [20,45,85,94,211]. Names should be easy to understand, consistent, distinctive and close to the domain language. The length of names does not have a significant impact on usability [13].	N06 the number of methods with the same prefix, as a measure for name distinctiveness
Code completion is a very important feature for looking up API elements. If something is not listed in the drop-down menu, most programmers won't believe it exists. On the other hand, having too many items in the drop-down menu makes it more difficult to find the required item. [45]	Note: The usability of an API depends not only on the structure of the API itself, but also on the IDE's code completion mechanism. It influences all N.. properties where lookup via code completion is supported.

Continued on next page

Table 3.1 – continued from previous page

API Usability Aspect	Potential Measurable Properties (MPs)	
Methods are very hard to find when they are placed in a class that has no direct relation to the code that the developer has already written. [184]	N07	the number of required classes
Method parameters that are self-explaining when reading code are better than ones that aren't, e.g. enum parameters are better than booleans. [45,94]	R03	parameters that are self-explaining are less complex than others (enums are the only reported self-explaining parameters)
The API should report usage errors as soon as possible, best at compile time [20]. For example, concrete data types or generics should be used instead of "object", if actually a certain type is required. APIs should be strongly typed whenever possible [45].	R04	parameters of undefined type (object, untyped collections) are more complex than others
	R05	concepts where compile time checking is limited or not possible, like XML configuration files and config strings, are more complex than others
Strings should not be used if a better type exists, since they are cumbersome and error-prone. [20]	R06	parameters of type string are more complex than others
Common usage scenarios should only require few namespaces/packages. Types that are often used together should reside in a single namespace if possible. [45]	N08	the number of required namespaces/packages
Functions with multiple consecutive parameters of the same type are especially difficult to use, because ordering mistakes are hard to find. [20]	N09	the number of consecutive method parameters with the same type
Parameters should be ordered consistently across methods. E.g. for a file API, the methods for copying and moving should have the same parameter ordering [20]. This is especially true for overloads – parameters with the same name should appear in the same position in all overloads [45].	N10	the number of different combinations of parameters with the same names in methods/overloads within a single class – e.g. the methods `copy(string fromPath, string toPath)` and `move(string toPath, string fromPath)` have two different combinations of the same parameters

Continued on next page

Table 3.1 – continued from previous page

API Usability Aspect	Potential Measurable Properties (MPs)
Exceptions should only be used to indicate exceptional conditions and incorrect API usage. Users should not be forced to use exceptions for control flow. If an exception needs to be thrown, already existing standard exceptions (e.g. ArgumentNullException) should be preferred, except when the error is very specific to the API. [20,45]	R07 exceptions are more complex than other classes (to motivate minimizing the use of exceptions) R08 API-specific exceptions are more complex than standard exceptions (to motivate the use of standard exceptions)
The abstraction level of an API should meet the expectations of the user. E.g. for network communication, TCP sockets have a very low abstraction level and are therefore hard to understand. [38, 150, 163]	N04 (see above; an API that uses the wrong abstraction level may require a larger number of concepts or more complex ones; e.g. compare low-level file streams to the simple .Net File API)
The factory pattern is more difficult to use than instantiation with a constructor [45, 58].	R09 Calling a factory method is more complex than instantiating a class. A factory method could be identified by checking if it is static and returns an instance of an API type.
Fields should not be public. This is a common view in both .Net (use properties instead) [45] and Java (use getters/setters) [188]. The main reason for this is that the field cannot be changed later in a backward-compatible way (e.g. to add validity checks when the value is set). The only exception are constants.	R10 public fields that are not constants are more complex than properties/getters/setters
Ambiguous method overloads (multiple overloads applicable to same objects) should be avoided. It may in some cases be better to use a different method name instead. [20]	N11 the number of ambiguous overloads (where at least one other overload exists that applies to the same combination of objects) N12 the number of overloads (less accurate, but easier to measure)

Continued on next page

Table 3.1 – continued from previous page

API Usability Aspect	Potential Measurable Properties (MPs)
Providing different API variants for beginners and experts can improve the overall learning curve [5, 142, 188]. For methods/constructors with many parameters, default overloads for simple use cases should be provided [45].	N04 (see above) R11 the number of method overloads has a less negative impact on usability than the number of method parameters
Developers will be uncomfortable with an API that exposes most, if not all, of it's functionality through annotations, due to the perceived lack of control afforded by annotations. Further, relationships or combinatorial effects between two or more annotations can be very confusing, if they are not clearly visible. [35, 36]	N13 the number of used annotations R12 the relation between used annotations and overall API elements should not exceed a certain level
A create-set-call pattern is easier to use than constructors with required parameters [182]. On the other hand, [150] did not find evidence for this. We believe it is strongly related to how a developer approaches an API. If a documentation is used (which was not the case in [182]), the negative impact of required parameters is likely limited.	R13 using a constructor with multiple parameters is more complex than using a parameterless constructor and then setting an equal number of properties

3.3.1 The User Factor

An important factor for usability, that was not dealt with in table 3.1, is clearly the user, which is the developer that uses the API to create software. For the API Concepts Framework a large influence of the user would of course be a disadvantage – if the user's experience strongly influenced the usability of an API, it would be very difficult to use the framework for getting a decision factor which API to use. Therefore the influence of the following aspects needs to be investigated:

- The developer's overall programming experience. A more experienced developer may be more self-confident, and/or find similarities to other APIs he/she already knows (also known as "transfer effects"), and therefore learn a new API easier. This could be evaluated by comparing the programming performance with the years of programming experience.
- The developer's experience with the problem domain. A developer who has already solved similar problems with other APIs may find similarities in the new API and therefore find it easier to use. This could be evaluated by comparing the programming performance with the developer's self-assessed level of domain knowledge.

- The developer's experience with a concrete API. A developer that is experienced with an API will perform better than one that has never used it.
- The developer's learning style. [37, 39] introduce the term *persona* for a stereotype with a specific learning style. E.g. a persona has certain expectations concerning an API's abstraction level. An API should be designed with these personas in mind. [182] reports that while there are differences in usability depending on the persona, in the end still the same API variant is preferred by all personas. Based on these results, we do not investigate this aspect further. To not enforce a purely explorative learning style on the developers (as done in most API usability studies [58, 182, 184]), which does not suite all personas, we also conduct studies where the developers can learn by using a documentation/tutorial.
- The used programming language. A Java developer may find different things easy or difficult to use than a .Net developer. We will take this aspect into account by investigating APIs for both Java and .Net.

3.3.2 Limitations

We identified some usability aspects that we were not able to cover sufficiently with measurable properties. The most important is *naming*, which is highly based on cognitive understanding. To measure naming, it would be necessary to understand the purpose of an API element, as well as have knowledge about the domain language, which is probably impossible for an automated framework. Another aspect is the API's *abstraction level*, which poses a similar problem. Further investigation will be required towards the impact of these aspects.

An aspect that we will deliberately not take into account are standard conventions, like naming conventions (e.g. upper/lowercase notation, certain prefixes and suffixes for class/interface names). Since such conventions are normally already checked by the IDE (e.g. using ReSharper[1] or FXCop[2] for .Net), it is unnecessary to integrate them into the API Concepts Framework.

[1]http://www.jetbrains.com/resharper/

[2]http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx

CHAPTER 4

Usability Study 1: Classes, Methods and Fields

We want to find out whether the measurable properties we identified in chapter 3 really have an influence on usability by conducting usability studies. Our goal is to do this as efficiently as possible, meaning we want to evaluate as many measurable properties in a single study as possible, but still without endangering the validity and significance of the results. We want to strengthen these results by applying statistical analysis methods, as well as a very fine-grained data evaluation method, which is described in more detail below.

We use an *A/B Test* [87] for the study, since it is the usability test that is most suitable for statistical evaluation. It involves creating two (or more) variants of an API, and letting users perform several tasks with both of them. The resulting performance times can then be compared statistically to find out which variant is better. Further, we use a *between-subjects design* for the study, meaning each participant is only tested with one of the APIs, to prevent a falsification of the data from cross-API learning effects (which is especially dangerous when the APIs are very similar, as it is the case in our study). Since not only the performance is important, but also to understand the reasons behind the user's behaviour, we additionally use *thinking aloud* [142], where the user is encouraged to talk about what he/she is currently doing while working on a task. While thinking aloud can influence the test (e.g. performance times) because the user thinks more about what he/she does, or gets distracted because of being not used to talking like that, the benefits are reportedly much greater than the risks. Finally, we present a *post-test questionnaire* to the user to find out more about his/her opinions concerning the used API.

The goal of the first study is to evaluate measurable properties of the most basic concepts of any API: classes, methods and fields. Since it is impossible to evaluate all measurable properties in a single study, even for only these three concepts, we concentrate on the measurable properties which we consider to be most important. Further, we want to investigate a usability aspect that poses one of the biggest threats to an automated framework: the influence of the user's experience. Table 4.1 shows a list of the properties and aspects to be evaluated (the labels refer to the definitions in section 3.3), and for each defines a way to statistically evaluate it by implementing and comparing two different API variants (called V1 and V2).

The following sections present the design and execution of the study, as well as the evaluation and interpretation of the study results. The results of this study were published in [168] and [169].

4.1 Design of the Study

4.1.1 Design of the API

As problem domain for the API we chose the domain of *file zipping* because most programmers are familiar with it and have basic knowledge of the domain language (zip, file, extract, ...), but haven't already worked with an API for zipping. This guarantees similar starting conditions for all test participants. To provide an API with meaningful methods, we investigated several existing ZIP APIs, like the DotNetZip[1] library.

We designed two API variants by first creating one variant, and then deriving the second variant from it, with changes according to the requirements listed in table 4.1. Table 4.2 gives a short overview of the differences between the two API variants. For API variant 2, the number of methods and classes was decreased to about 1/3 of API variant 1. We implemented our APIs for both Java and .Net as equally as possible, but in both cases following platform-specific conventions (e.g. upper/lower case, getters/setters in Java, properties in .Net).

Programmers had to solve 8 different tasks within 45-60 minutes time, with each task involving one specific class and method that needed to be found and used. Listing 4.1 shows the source code for each task. The following tasks had to be solved:

1. Zipping a single file
2. Extracting all files from an existing zip file
3. Extracting a single file from an existing zip file
4. Printing the content of a zip file to the console
5. Updating the content of a file within a zip file
6. Removing a file from an existing zip file
7. Zipping a file with a password
8. Zipping a stream in memory

Each task basically consisted of the following actions which can be evaluated separately: Find and instantiate the correct class, then find and call one or two methods. For all 8 tasks the same class (`ZipFile`) needed to be used, allowing us to evaluate the learning effect. Further, presenting the user so many different tasks allowed us to evaluate method calls in different situations, and to get a larger number of data for statistical evaluation.

[1] http://dotnetzip.codeplex.com/

Table 4.1: Properties and Aspects to be Evaluated in the Usability Study

Property / Aspect	**Way to Measure**
Has the **number of classes in the same package** an influence on finding a class? How does it compare to the overall number of classes in the API? N01 N02 R02	V1: 30 classes in the package V2: 10 classes in the package, other classes placed in sub-packages
Is instantiation with a **constructor** faster than with a **factory**? (This question is also covered in detail by [58], but we cannot extract usable performance times from this paper, so we will nevertheless investigate this on our own) R09	V1: Instantiation with a constructor V2: Instantiation with a static method which is placed in the same class
Does the **number of other members in the class** have an influence on finding a method? N03	V1: class with ~100 members (many existing APIs have a large number of members) V2: only one third of the members of V1
How big is influence of the **number of method parameters**? N05	Methods with different numbers of parameters (0-4) in both API variants
Does the **number of overloads** have an influence on finding the most suitable overload? Is it better to have a higher number of overloads, or a higher number of method parameters? N12 R11	V1: a method with 8 overloads, an overload with 2 parameters needs to be used V2: a method with 4 overloads, an overload with 4 parameters needs to be used
Does the **number of methods with the same prefix** have an influence on finding a method? Does it have a greater influence than the overall number of methods? N06	V1: 30 choices with the same prefix V2: 5-10 choices with the same prefix The best suitable method at a position in the middle for both variants
Are properties in .Net equal to getters/setters concerning usability?	V1: setter method V2: property (in .Net only)
Does the IDE's **code completion** mechanism have an impact on usability?	Compare between users of Eclipse and Visual Studio: method search performance, success rate for finding the optimal method/overload
Does the **developer's experience** have an influence on performance?	Check for correlation between performance and years of programming experience
Does the **developer's domain knowledge** have an influence on performance?	Check for correlation between performance and the developer's self-assessed level of domain knowledge
How large is the performance increase after a developer has learned how to use an API, in other words the **learning effect**?	Use the same API elements in multiple consecutive tasks and check how much the performance increases.

Table 4.2: Comparison of API variants

	API 1	**API 2**
Classes in package	33	8
Methods in ZipFile class	30	10
Method overloads in ZipFile class	50	20
Getter/Setter pairs in ZipFile class	33	12
Instantiation of ZipFile class by	Constructor	Factory

As an example, the actions needed to finish task 1 (as shown in listing 4.1) were as follows: First the programmer had to find and instantiate the class `ZipFile` (allows creating the zip file), then find and call the method `addFile` (adds a file to the zip file). Finally, the method `save` had to be called (saves the zip file to the file system). For each task there was one optimal way for reaching a solution, as well as a number of other possible solutions requiring additional programming effort.

All tasks were designed as unit tests (using jUnit in Java, and NUnit in .Net), so that programmers could easily execute their code and see if the results are correct. The structure of a single task is shown in listing 4.2. On the top, a short textual description of the task was given. To distract the user as little as possible, code for preparation in the beginning (e.g. remove or create files) and for assertions in the end (e.g. check correctness of created zip file) was extracted into a different class (`UsabilityTestRunHelper`). Further, all information that was required in code, like filenames that were needed by the user to solve the task, were given as predefined variables. The place where the user should write his/her code was marked with “code start” and “code end”.

4.1.2 Design of the Post-Test Questionnaire

A post-test questionnaire was used to evaluate the opinion of the users about the tested API. To stick to the native language of the test users, the questionnaire was kept in German. [142] suggests two different scales that are used for usability questionnaires. One is the *Likert scale*, where the user is asked to rate a certain statement (e.g. “The API was easy to learn and use”) on a 5- or 7-point scale, ranging from strong agreement to strong disagreement. The other variant is the differential scale, where two opposing terms (e.g. simple ... complex) are given, and the user needs to give a rating on a 5- or 7-point scale between the two. We tested out both variants, and after obtaining some opinions about them, decided to use the differential scale, because it was easier to understand. Figure 4.1 shows the questions as they were presented on the questionnaire. The complete questionnaire can be found in appendix A.

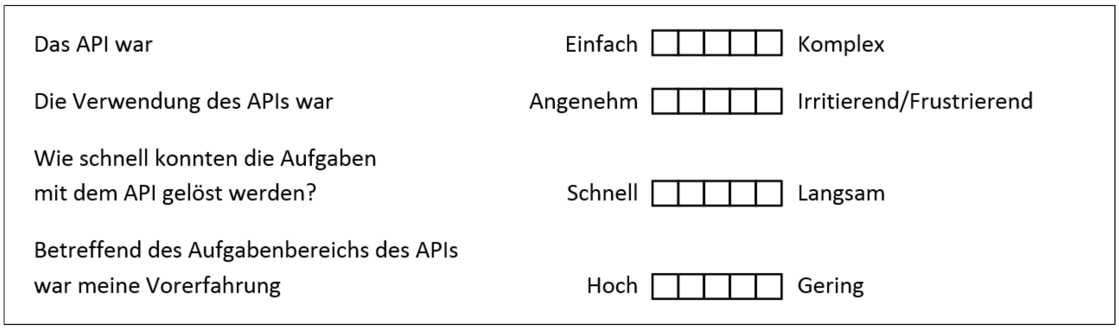
In addition to these question, users were asked for their age, years of programming experience, as well as what they especially liked or disliked about the API. Many users were unsure about what to count as years of programming experience, so we particularly asked them to rate only the time since they had been doing “real” projects (basically excluding school programming education).

```
//Instantiation for all tasks for creating a zip file
ZipFile zip = new ZipFile(); //API 1 (constructor)
ZipFile zip = ZipFile.CreateNew(); //API 2 (factory method)

//Instantiation for all tasks for reading a zip file
ZipFile zip = new ZipFile(filename); //API 1 (constructor)
ZipFile zip = ZipFile.Read(filename); //API 2 (factory method)

//Task 1
zip.AddFile(fileToZip);
zip.Save(zippedFile);

//Task 2
zip.ExtractAll(unzipPath);

//Task 3
zip.Get(fileToUnzip).Extract();

//Task 4
foreach (ZipEntry e in zip.Entries)
{
   Console.WriteLine(e.FileName);
}

//Task 5
zip.UpdateEntry(fileToUpdate, contentToUpdate,
   null, UpdateAction.Overwrite); //API 1 (more params)
zip.UpdateEntry(fileToUpdate, contentToUpdate); //API 2 (less params)
zip.Save();

//Task 6
zip.RemoveEntry(fileToRemove);
zip.Save();

//Task 7
zip.SetPassword(password); //setter variant (API 1 + Java API 2)
zip.Password = password; //property variant (.Net API 2)
zip.AddFile(fileToZip);
zip.Save(zippedFile);

//Task 8
zip.AddEntry("dummyName", inputStream);
zip.Save(outputStream);
```

Listing 4.1: Example solutions for the 8 tasks (.Net variant)

```
/// <summary>
/// Zip a single file.
/// </summary>
[Test]
public void Zip_Single_File()
{
   UsabilityTestRunHelper.PrepareTest_Zip_Single_File();

   string fileToZip = "test.txt";
   string zippedFile = "Archive.zip";

   //code start

   //code end

   UsabilityTestRunHelper.CheckResults_Zip_Single_File();
}
```

Listing 4.2: Unittest-based task structure for study 1



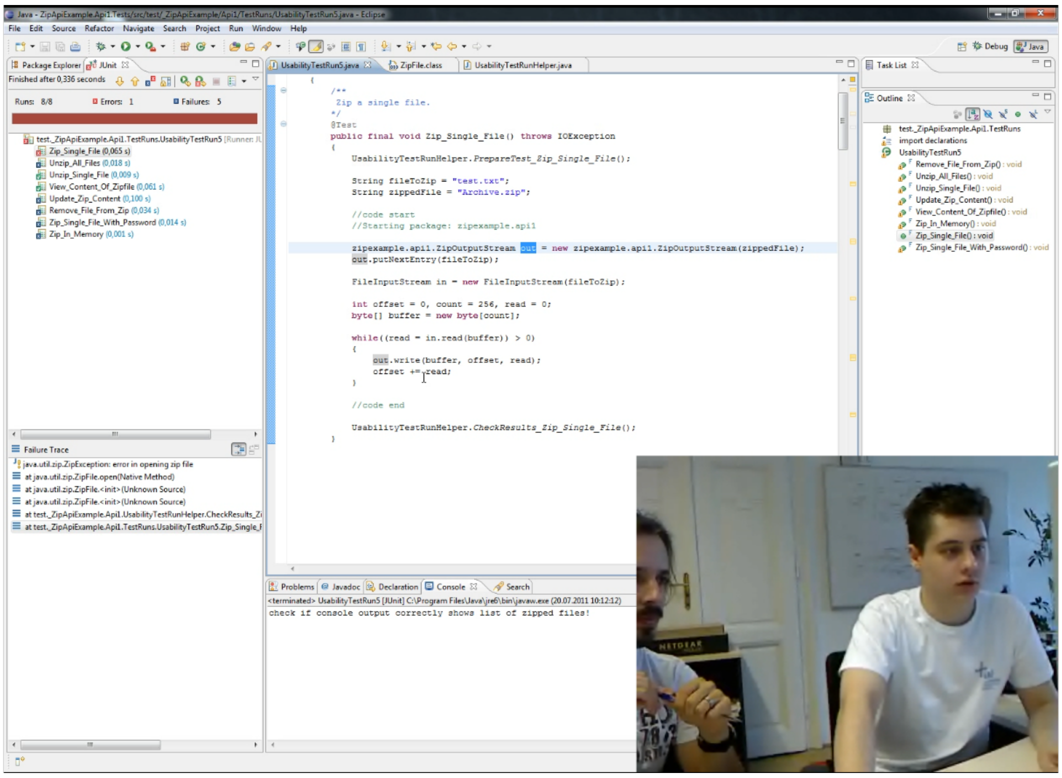
Das API war	Einfach	☐☐☐☐☐	Komplex
Die Verwendung des APIs war	Angenehm	☐☐☐☐☐	Irritierend/Frustrierend
Wie schnell konnten die Aufgaben mit dem API gelöst werden?	Schnell	☐☐☐☐☐	Langsam
Betreffend des Aufgabenbereichs des APIs war meine Vorerfahrung	Hoch	☐☐☐☐☐	Gering

Figure 4.1: Excerpt of the questionnaire from study 1 (in German)

4.1.3 Participants and Environment

Our study included 20 programmers, who were recruited personally from students and project assistants from the Institute of Computer Languages at TU Vienna, as well as from the company partner of the AgiLog project pcsysteme.at. 10 were working with Java and 10 with .Net (C#). Java programmers used Eclipse as IDE, .Net programmers used Visual Studio (5 with additionally the ReSharper addon installed). All of them had between 2 and 10 years of programming experience and were between 20 and 30 years old. Programmers were equally split between the two API variants (5 Java and 5 .Net programmers per variant). The study involved both programmers with academic and industrial background, who had experience with a variety of different APIs.

Programmers were learning purely by exploring the API, and additionally had the Javadoc / XML documentation available as a learning resource (viewable e.g. using the code completion functions). While this forced an exploratory learning approach on the users, it had several big advantages: it removed any validity threat that could be connected to the documentation, it



```
Finished after 0,336 seconds
Runs: 8/8    Errors: 1    Failures: 5

test._ZipApiExample.Api1.TestRuns.UsabilityTestRun5 [Runner: JU
    Zip_Single_File (0,065 s)
    Unzip_All_Files (0,018 s)
    Unzip_Single_File (0,009 s)
    View_Content_Of_Zipfile (0,061 s)
    Update_Zip_Content (0,100 s)
    Remove_File_From_Zip (0,034 s)
    Zip_Single_File_With_Password (0,014 s)
    Zip_In_Memory (0,001 s)
```

```
Failure Trace
java.util.zip.ZipException: error in opening zip file
at java.util.zip.ZipFile.open(Native Method)
at java.util.zip.ZipFile.<init>(Unknown Source)
at java.util.zip.ZipFile.<init>(Unknown Source)
at test._ZipApiExample.Api1.UsabilityTestRunHelper.CheckResults_Zi
at test._ZipApiExample.Api1.TestRuns.UsabilityTestRun5.Zip_Single_F
```

```java
	{
		/**
		 Zip a single file.
		*/
		@Test
		public final void Zip_Single_File() throws IOException
		{
			UsabilityTestRunHelper.PrepareTest_Zip_Single_File();

			String fileToZip = "test.txt";
			String zippedFile = "Archive.zip";

			//code start
			//Starting package: zipexample.api1

			zipexample.api1.ZipOutputStream out = new zipexample.api1.ZipOutputStream(zippedFile);
			out.putNextEntry(fileToZip);

			FileInputStream in = new FileInputStream(fileToZip);

			int offset = 0, count = 256, read = 0;
			byte[] buffer = new byte[count];

			while((read = in.read(buffer)) > 0)
			{
				out.write(buffer, offset, read);
				offset += read;
			}

			//code end

			UsabilityTestRunHelper.CheckResults_Zip_Single_File();
		}
```

```
<terminated> UsabilityTestRun5 [JUnit] C:\Program Files\Java\jre6\bin\javaw.exe (20.07.2011 10:12:12)
check if console output correctly shows list of zipped files!
```

Java - ZipApiExample.Api1.Tests/src/test/_ZipApiExample/Api1/TestRuns/UsabilityTestRun5.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
JUnit
UsabilityTestRun5.java
ZipFile.class
UsabilityTestRunHelper.java
Task List
Outline
Problems
Javadoc
Declaration
Console
Search
Debug
Java

Figure 4.2: Example screenshot of the video evaluation from study 1

forced the programmers to use code completion and by that better showed how this feature influences usability, and it reduced the study preparation effort.

All programmers were recorded using the screen capturing software *Microsoft Expression Encoder*, capturing the screen and audio, as well as a webcam video. Figure 4.2 shows an example screenshot from a captured video. In the video file that was rendered for evaluation the webcam video was placed in the lower left corner, because there was no other essential information for the evaluator in this corner of the screen. A supervisor was sitting next to the programmer to explain each task, and if a programmer was stuck, cancel the task and let him/her continue with the next one.

Other than explaining the task, the experimenter was not allowed to help the user during the test. Only if the experimenter really believed the user to be at a dead end, from which he cannot recover by himself, the experimenter was allowed to give a hint to put the user on the right path again to avoid frustration (which could also have had a negative impact on the following test tasks). Nevertheless, in such a case the current task was of course rated as unsuccessful.

4.1.4 Study Execution

The following stages were gone through for each test run. This structure loosely follows the suggestions given in [142].

1. **Preparation:**
    - Bring the computer system into the starting state before the user shows up. Depending on which programming language the user will be using, start either Eclipse or Visual Studio, and prepare an empty test template into which the user will write the code. Check if everything is set up correctly by testing the first task.
    - Start the screen capturing software and check if webcam and sound are working correctly.
    - Prepare the questionnaire and anything else needed to take notes, etc.
2. **Introduction:**
    - Emphasize that it is the system that is being tested, not the user.
    - Let users know that they can stop at any time.
    - Make sure that you have answered all the user's questions before proceeding.
    - Explain how the test will be carried out.
    - Shortly describe the purpose of the API that will be tested.
    - Ask for the user's kind approval to record him/her via webcam.
3. **The test itself:**
    - Hand out the test tasks one at a time.
    - Keep a relaxed atmosphere in the test room, serve drinks and/or have breaks.
    - Avoid disruptions.
    - Take notes about any interesting observations.
    - Never indicate in any way that the user is making mistakes or is too slow.
    - Try to give the user an early success experience (go from easy to difficult tasks).
    - Silently observe the test user during a task (no helping interaction).
    - Encourage the user to think aloud, but don't enforce it (the user may get distracted, leading to a falsification of the performance data).
    - Don't ask any questions, even if the user does something that is not understandable. But do collect such questions to ask them later in the debriefing phase.
4. **Debriefing:**
    - Before anything else, let the user fill out the questionnaire.
    - After that answer all additional questions the user may have, write down any comments of interest that are given by the user.
    - Ask the user any questions that may have come up during the test.
    - End by thanking the user for participation.

Table 4.3: Gathering and preparing the study data - step 1

Start Time	Action	Searched by
00:00	Search for Class	List
00:20	Instantiate ZipFile	
00:37	Search for Method (AddFile)	List
00:43	Use Method (AddFile)	
00:52	Search for Method (Save)	List
01:39	Use Method (Save)	
01:44	Run Test	
01:46	Finished - OK	

4.1.5 Gathering and Preparing the Study Data

From the captured video data, we extracted times for specific steps that programmers had to do in each task. These steps included searching for a class, instantiating the class, searching for a method, calling the method, running a test, and searching for errors. Splitting the performance data into small parts removes unnecessary noise (like the time programmers need to execute the unit tests, which is irrelevant for our purpose) and by that makes the data more suitable for statistical interpretation. Additionally, we recorded the number of successful tasks (when the experimenter had to give a hint, the task was rated as unsuccessful), as well as the number of times where the user expressed clear frustration or joy. Since the latter happened very infrequently, we found the data to be unrepresentative and refrained from using it for further evaluation.

Further, for every programmer and task we evaluated whether the optimal solution was used (optimal class, method and overload). When programmers were searching for methods, we observed that the time needed strongly depended on how the search was performed. While some programmers preferred searching by entering a certain prefix or hint and then looking at the remaining choices, others preferred scrolling through the whole list of classes or methods. We therefore evaluated for every search whether it was done *by hint* or *by list*.

The evaluation of the video data was done in several consecutive steps. The first step was simply splitting the data into a number of consecutive time frames and assigning activities to them, as well as checking how search was performed (by hint or by list). This was done by watching the video, and whenever the user changed his/her current activity, pause it and record the current time in an Excel sheet. This step was also used to filter disruptions (e.g. when the user asked questions because something about the task goal was unclear), simply by recording an empty activity for that time. An example for this evaluation step is shown in table 4.3 for task 1. This step was repeated for each user and task, so with 20 users and 8 tasks per user, 160 videos were evaluated. All tasks of a single user were recorded in a single Excel file.

In the second evaluation step, the values for all parameters that were required for statistical evaluation were extracted from the step 1 results, and put into a standardized format. The results of this step are shown in table 4.4. Most important in this step was the calculation of the durations of certain actions, which was done by substracting the start time of the action from the start time of the following action. Further, multiple time frames belonging to the same action were

Table 4.4: Gathering and preparing the study data - step 2

Parameter	**Value**
Overall Time	00:01:46
Search for Class	00:00:20
Instantiate ZipFile	00:00:17
Search for Method	00:00:06
Use Method	00:00:09
Search for Error	00:00:00
Run Test	00:00:02
Finished	yes
Searched by hint	no
Searched by list	yes
Used expected class	yes
Used expected method	yes
Used expected overload	yes

summed up into a single value (e.g. when the user searched for a method, then did something else, and then continued searching). The whole calculation of this step was done automatically using the `IF` and `SUMIF` functions in Excel. Only the data concerning the usage of the expected class/method/overload was added by hand (last three lines in table 4.4).

In the third and final step, the data of a single user was brought into a compact one-line format, which was then copied over into an accumulated Excel file containing the data of all tests/users. To make this as simple as possible, each single-user excel file contained a template for the one-line format which was automatically filled out and could simply be copied by using Excel's "insert values" (instead of formulas) function. An excerpt of this result (only showing a part of the users, and only task 1) is presented in table 4.5.

In addition to the video evaluation data, we also integrated the results of the post-test questionnaire into this final Excel file. We therefore converted the differential scale of the questionnaire into a 5-point rating for the values simplicity, satisfaction, speed and domain knowledge. Further, for each participant it was recorded whether he/she felt that the API's number of classes, methods or overloads had had any negative impact on his/her performance.

Finally, we converted this resulting Excel file into a CSV file, which could be imported into the statistical computing tool *R* for statistical evaluation of the data. With 13+ values per task and user, as well as the questionnaire data, about 2500 independent values overall were collected this way.

4.2 Study Results

Most of our performance results showed a significant floor effect, i.e. the data was not normally distributed. For statistical analysis we therefore used the *Wilcoxon rank-sum test* (further called *Wilcoxon* in short), which is a non-parametric statistical hypothesis test for assessing whether

Table 4.5: Gathering and preparing the study data - step 3

Test Number	API Version	Language	IDE	Overall Time	Time to Search for Class	Time to Instantiate Class	Time to Search for Method	Time to Use Method	Used exp. Class	Used exp. Method	Used exp. Overload	Searched by Hint	Searched by List	Success
1	2	.Net	VS+R#	03:15	00:30	00:43	01:28	00:09	x	x	x		x	x
2	1	.Net	VS+R#	04:14	01:14	01:21	00:11	00:09	x	x	x	x		x
3	1	Java	Eclipse	03:16	01:50	00:16	00:23	00:09	x				x	x
4	2	.Net	VS	02:44	00:26	00:44	00:05	00:15	x	x	x	x		x
5	2	Java	Eclipse	04:56	00:51	00:53	01:41	00:06	x	x	x		x	x
6	1	Java	Eclipse	04:02	01:23	00:58	00:30	00:05	x	x	x	x		x
7	2	Java	Eclipse	03:12	00:40	00:41	00:32	00:48	x			x	x	x
8	1	Java	Eclipse	01:46	00:20	00:17	00:06	00:09	x				x	x
9	...													

one of two samples of independent observations tends to have larger values than the other. For every evaluation, the sample sizes (n_1, n_2) are given, as well as the p-value. If the p-value is below 0.05, it means that there is a significant difference between the two samples (the null-hypothesis is rejected).

All statistics and plots were created using *R*, with additional use of the plotting add-in *ggplot2*. The scripts that have been used to create the R plots and statistics can be found in appendix C.

The following sub sections describe the results of the study in detail.

4.2.1 Searching for Classes

In all 8 tasks, programmers had to use the `ZipFile` class. In API variant 1 only a single package existed with 33 classes in it. API 2 contained the same number of classes, but they were split up into several sub packages, with only 8 classes remaining in the main package, which is about the number of items fitting in the IDEs' code completion windows.

When searching for a suitable class, about half of the programmers first tried to get an overview of the classes in the main package, while the other half looked directly for classes with specific names (mostly ones starting with "Zip"). After the first or second task, programmers showed a strong learning effect, so that most of the time they used the `ZipFile` class right away and didn't need to search any more. To compare the search times we therefore only took the times from the first task.

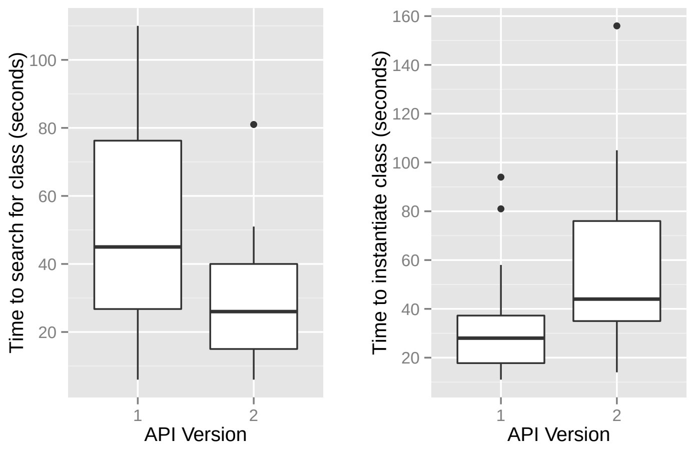


Figure 4.3: Boxplots: (a) times to search for class in API 1 (more classes) and 2 (fewer classes), (b) instantiation times with API 1 (constructor) and 2 (static factory method)

The search times for API 1 and 2 are shown in Figure 4.3(a): For all cases where programmers found and used the class `ZipFile`, API 1 shows a median search time of 45 seconds compared to API 2 with a median time of 26 seconds. The Wilcoxon test indicates a high probability for a significant difference between search times of the two APIs (n_1=9, n_2=8, p=0.114). This shows that even such a simple package restructuring as it has been done in API 2 can improve performance.

When asking programmers if they were bothered by a large number of classes, 7 out of 10 using API 1 were bothered by it, but only 1 out of 10 using API 2. So, programmers clearly recognized this disadvantage of API 1. A typical statement was that "there seem to be many classes that are internal, and don't need to be seen by a programmer that is using this API" (which was a false perception by the programmers – all the classes actually were a meaningful part of the public API, but were just not used in the tasks that the programmers had to solve).

When asked what to do about this problem, a common statement was that the programmers would have wished for "a hint for which are the most important classes of the API", which would have prevented the large number of classes from bothering them. Only 2 programmers had the idea that a simple restructuring as it was done with API 2 would help. This shows that, although many programmers recognized that there was a usability problem, most of them didn't think about how the API itself could be changed.

4.2.2 Instantiating Classes

Constructor vs Factory

We compared two different instantiation variants. API 1 contained a constructor for instantiation, API 2 used static factory methods:

```
ZipFile zipFile = new ZipFile("file.zip");
ZipFile zipFile = ZipFile.createNew("file.zip");
```

Although a similar evaluation has already been done in [58], we see our evaluation as a useful complement to this work, since it shows two major differences: First, while in [58] all test cases present the factory pattern by using an additional factory class (e.g. `SslSocket` + `SslSocketFactory`), we analyze a case where the static methods are contained in the actually instantiated class. Second, while [58] analyzes the overall task completion times, we apply a more detailed measurement by extracting only the time for instantiation.

Again, programmers showed a strong learning effect after the first two tasks. To see if there is a difference both when programmers are not yet accustomed to the API, as well as when they are already familiar with it, we analyzed these two cases separately. We first evaluated the instantiation times of tasks 1 and 2 only, where programmers were not yet accustomed to the API. A comparison of the times where the `ZipFile` class was successfully instantiated is shown in Figure 4.3(b): API 1 (instantiation with constructor) has a median time of 25 seconds, compared to API 2 (instantiation with static method) with a median time of 44 seconds. There is a significant difference between the instantiation times of the two APIs (Wilcoxon: n_1=n_2=13, p=0.019), with times being about twice as high when using static methods.

When comparing the instantiation times over all tasks and by that considering the learning effect, the median times are 15 seconds for API 1 and 21 seconds for API 2. There is a significant difference also in this case (Wiloxon: n_1=n_2=43, p=0.0001). The difference in performance is not as large as when using the API for the first time, but there still about 30% more time required with static methods.

Surprisingly, with both API variants about half of the programmers had problems when instantiating the `ZipFile` class for the first time. With API 1, 4 programmers had problems because they expected that the `ZipFile` class would contain static methods, and were questioning their choice of class when they discovered that there were no fitting static methods. With API 2, 6 programmers had instantiation problems because they tried to instantiate the `ZipFile` class by constructor, which was private in this case. As also reported in [58], programmers got very confused in this case since the IDE showed the private constructors in the code completion window, and the error message didn't give a clear hint to what the problem was.

After the test, we asked programmers which type of instantiation they preferred (by constructor or by static method). 9 programmers answered that they preferred constructors, only 4 preferred static methods, 7 answered that both variants were equal for them.

Factory: .Net vs Java

For the factory instantiation variant, we additionally compared whether .Net users were slower than Java users. This suspicion arose because a prominent Java guidelines book [188] suggests to always prefer factories, while .Net guidelines [45] suggest the opposite. If this was the case, Java users would have been able to find the factory method faster and also to use them better. According to the results of our study, this is not the case, as shown in figure 4.4. There is no significant difference between the search and instantiation times for Java and .Net users (Wilcoxon: n_1=5, n_2=8, p=0.284).

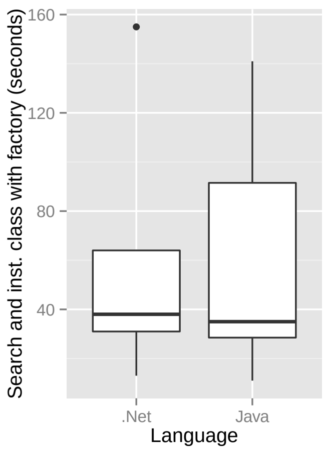


Figure 4.4: Boxplot: Instantiation times with a factory in .Net vs Java

4.2.3 Learning Effect

To assess the degree of performance improvement when programmers have learned how to use the `ZipFile` class, we compared the search and instantiation times for each task, as shown in Figure 4.5. For task 1, the median time to search and instantiate the `ZipFile` class is 80.5 seconds. About half of the time is needed for searching, the other half for instantiation. Over all tasks, the median time is 19 seconds, which is a time reduction of about 75%. Due to the learning effect the search time goes to zero, and the instantiation time is cut by half. Task 8 was omitted from this evaluation because programmers tended not to use the `ZipFile` class for it.

Figure 4.5 also shows that after a certain degree of learning, performance doesn't improve any more. In case of instantiation of the `ZipFile` class, programmers achieved the maximum degree of learning with the third usage (task 3).

The degree of learning (the number of usages until the maximum learning effect is reached, as well as the improvement in performance) may of course be different for more or less complex API concepts. We expect that a more complex API concept shows stronger learning effects than a simple one. Further studies will be needed to analyze learning effects in more detail.

4.2.4 Searching for Methods

Influence of the Number of Members in the Class

We compared the times programmers needed when searching for a method to find out if the number of class members has an influence on performance. For this, we compared API variant 1 with 146 non-static members (methods, overloads, getters/setters) to API variant 2 with 54 members in the `ZipFile` class. The method search performance data from all 8 tasks was used.

We observed that programmers showed noticeable differences in search behaviour. The most significant difference we observed is that some programmers tend to search mainly *by list*,

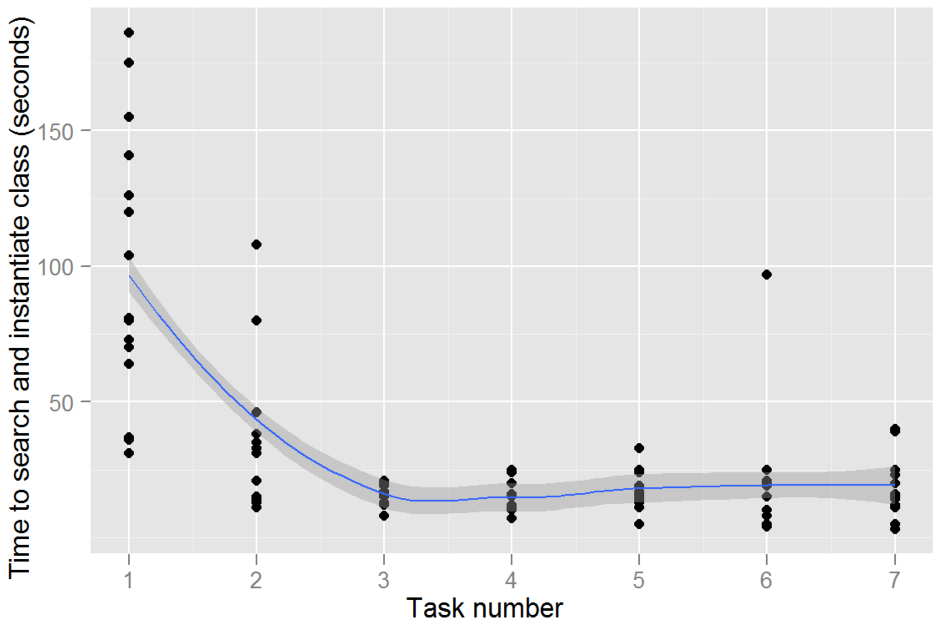


Figure 4.5: API learning effect: Search and instantiation times compared to the task number.

meaning they manually scroll through the list of members to find a suitable method, while others prefer to search *by hint*, meaning they type in prefixes to limit the available choices and don't scroll through the whole member list. For example, during task 1, where programmers were searching for a method to add a file to a zip file (also see Listing 4.1), they would enter the prefix "add" which would lead to the code completion window only showing the method names starting with this prefix. In general, programmers searching by hint were much faster than when searching by list, with a median time of 5 seconds for searching by hint compared to 22 seconds for searching by list. From 98 observed searches, programmers searched 48 times purely by list and 43 times purely by hint. 7 times they searched first by hint and then by list because they hadn't found a suitable method with the used hints. So, the searches were split about equally among the two search methods.

The second difference in search behaviour was that while most programmers preferred to use the code completion window, some preferred to use the IDE's external library viewer instead (Eclipse: "Referenced Libraries" in package explorer, Visual Studio: object browser). The latter tended to be more time-expensive due to the programmer having to switch to another window to explore the API's members. Only 4 of the 20 programmers (3 with Eclipse, 1 with Visual Studio) preferred using the external library viewer over the code completion window.

To see if there is a difference between API 1 and 2, we compared all cases where the optimal method was used. A boxplot of the comparison can be found in Figure 4.6(a). Although the search times tend to be higher for API 1, the difference between the two API versions is surprisingly small. The median search times are 14.5 seconds for API 1 and 12 seconds for API 2. There is no significant difference between the two API variants (Wilcoxon: n_1=32, n_2=27, p=0.226).

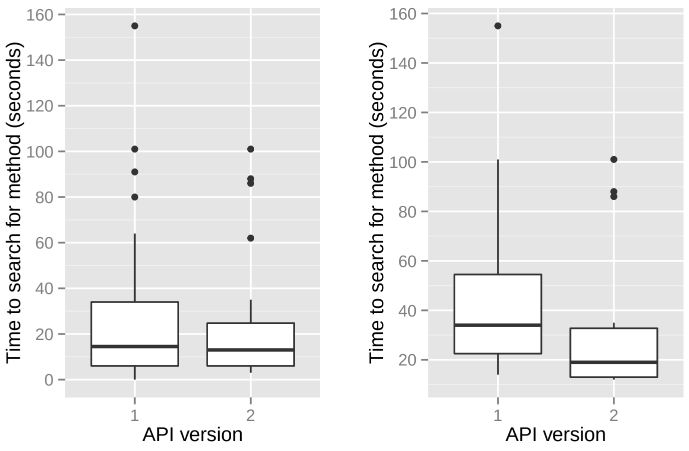


Figure 4.6: Boxplots comparing the times to search for methods for APIs 1 (more methods) and 2 (fewer methods): (a) all search times, (b) times where the programmer searched by list.

One of the main reasons for this can be found when taking the different search behaviours into account: When searching by hint, the overall number of methods is mostly irrelevant, because the programmer only sees the methods fitting to the searched prefix.

Because of that, we refined our comparison, taking into account only the times where programmers searched by list (see Figure 4.6(b)). Additionally, we only measured the cases where programmers really had to scroll through the list, meaning the wanted method was not among the first few in the list. The median search times in this case are 34 seconds for API 1 and 19 seconds for API 2. The times for API 1 are significantly higher (Wilcoxon: n_1=15, n_2=14, p=0.048). This shows that a different number of members really has an impact on search performance, although the difference is smaller than expected.

When asking programmers if they were bothered by a large number of methods, 4 out of 10 for API 1 and 3 out of 10 for API 2 felt that the number of methods had a negative impact. A majority of the programmers answered that it didn't bother them, and that a high number of methods just indicates that the API has many functions. We were surprised by this fact, because we expected that programmers would rather see this as an indicator of bad API design, because the functionality could also have been split among several classes. Only one programmer expressed the idea that splitting the functionality of the `ZipFile` class could have helped for a more well-arranged API.

Influence of the IDE's Code Completion Mechanism – Success Rate

As described in section 2.2.2, there are differences in the code completion mechanisms of different IDEs, especially between ones for Java and .Net. To check whether this has an impact on usability, we analyzed how often the optimal method and overload has been found when using the code completion window (success rate), and if there are any differences in performance. We therefore compare the three IDEs where we found the most differences in code comple-

Table 4.6: Number of optimal methods found

IDE	found	not found	percentage
Eclipse	56	10	85%
Visual Studio	30	2	94%
VS + ReSharper	31	1	97%

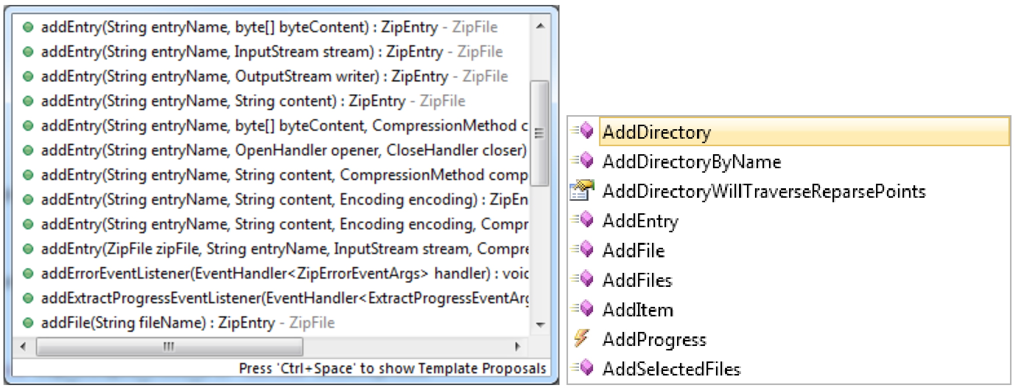
tion mechanisms (see section 2.2.2), namely Eclipse, Visual Studio, and Visual Studio with the ReSharper addon.

For all cases where the correct class was used, we evaluated whether programmers used the optimal method. In Eclipse, programmers found and used the optimal method in 56 out of 66 cases. With Visual Studio, the results with and without ReSharper are very similar, which is not surprising since the first step of the code completion works equally in both cases. Combining these two variants, the optimal method was found in 61 of 64 cases. Table 4.6 summarizes these results. To analyze the data we used *Fisher's exact test* [66], which is a statistical significance test used for the analysis of contingency tables, and has the advantage of providing exact results even for small sample sizes. A comparison of Eclipse with the two Visual Studio variants shows a p-value of 0.0766, so there is a high probability that the difference between the IDEs is significant.

For cases where the optimal method was not found we tried to find more evidence why exactly the users were not able to find them. From the 8 tasks that the programmers had to accomplish, one stands out the most, where from the 10 Eclipse users only 5 were able to find the optimal method, while all Visual Studio users found the method without problems. The goal of this task was creating a zip file by compressing a single file. To do that, users had to find the method `addFile` in the class `ZipFile`. In addition to this method, especially API variant 1 contained many methods with the same prefix, like `addDirectory` and `addEntry`, which also had multiple overloads. In most cases users searched for methods with the prefix "add". Figure 4.7 shows a comparison of the corresponding code completion results in Eclipse and Visual Studio. While Visual Studio displays no overloads and therefore shows the desired method at position 5, in Eclipse due to the large number of overloads the method is shown at position 18.

Since users typically read the code completion window top down, in Eclipse they often began to closer investigate the overloads of `addEntry` and didn't read further down. Methods of choice were for example the two `addEntry` overloads that can be seen at the top of the code completion window shown in Figure 4.7(a), one taking an entry name and a byte array, the other one an entry name and an input stream. To use these methods, users first had to manually read the file content, e.g. by using a `FileInputStream`, which resulted in a more complex code. The code in these cases also tended to be more error prone than with the optimal method – two of the users that used `addEntry` didn't close the stream, so the file remained locked.

For all cases where the optimal method was found, we evaluated whether programmers used the optimal overload. The results are shown in Table 4.7. This time both Eclipse and ReSharper show a high number of optimal finds (over 95%), while only 87% of the Visual Studio users were able to find the optimal overload. This could be related to the difference of displaying


```
addEntry(String entryName, byte[] byteContent) : ZipEntry - ZipFile
addEntry(String entryName, InputStream stream) : ZipEntry - ZipFile
addEntry(String entryName, OutputStream writer) : ZipEntry - ZipFile
addEntry(String entryName, String content) : ZipEntry - ZipFile
addEntry(String entryName, byte[] byteContent, CompressionMethod c
addEntry(String entryName, OpenHandler opener, CloseHandler closer)
addEntry(String entryName, String content, CompressionMethod comp
addEntry(String entryName, String content, Encoding encoding) : ZipEn
addEntry(String entryName, String content, Encoding encoding, Compr
addEntry(ZipFile zipFile, String entryName, InputStream stream, Compre
addErrorEventListener(EventHandler<ZipErrorEventArgs> handler) : voic
addExtractProgressEventListener(EventHandler<ExtractProgressEventAr
addFile(String fileName) : ZipEntry - ZipFile
Press 'Ctrl+Space' to show Template Proposals
```

```
AddDirectory
AddDirectoryByName
AddDirectoryWillTraverseReparsePoints
AddEntry
AddFile
AddFiles
AddItem
AddProgress
AddSelectedFiles
```

Figure 4.7: Code completion results for the prefix *add*: (a) Eclipse, (b) Visual Studio.

Table 4.7: Number of optimal overloads found

IDE	found	not found	percentage
Eclipse	54	2	96%
Visual Studio	26	4	87%
VS + ReSharper	30	1	97%

overloads: Both Eclipse and ReSharper display the overloads as a list, which makes it easy to get an overview of the available overloads just by taking a short look. On the other hand, Visual Studio only displays one overload at a time and shows a hint about the number of overloads (e.g. “2 of 4”, see Figure 2.5(2)).

Again we analyzed the data for statistical significance with *Fisher’s exact test*: We combined the results of Eclipse and ReSharper and compared them to Visual Studio. The resulting p-value of 0.0706 again indicates a high probability that this difference is significant.

A closer investigation of why the optimal overloads were not found shows that this was especially the case when the optimal overload was placed somewhere further down (e.g. the 4th or 5th overload in the list). In Visual Studio, users would sometimes just not look through all of the overloads, but only the first two or three, or sometimes they wouldn’t even recognize at first sight that the method had overloads at all.

After showing the methods to the programmers that they oversaw, some of them could not explain why they didn’t see the methods (and were actually surprised themselves about this fact), or scroll down further so that the method would have become visible. One Visual Studio user said that he would have expected the most relevant method overload to be on top (which wasn’t the case since overloads are sorted by parameter count and alphabetically, starting with the first parameter’s type). This could also be an indication that, while the alphabetical ordering of methods is clear, it may not be so clear by which criteria the overloads are ordered. After all, in this case the programmers are not searching for a specific name, but rather for a specific combination of parameter types.

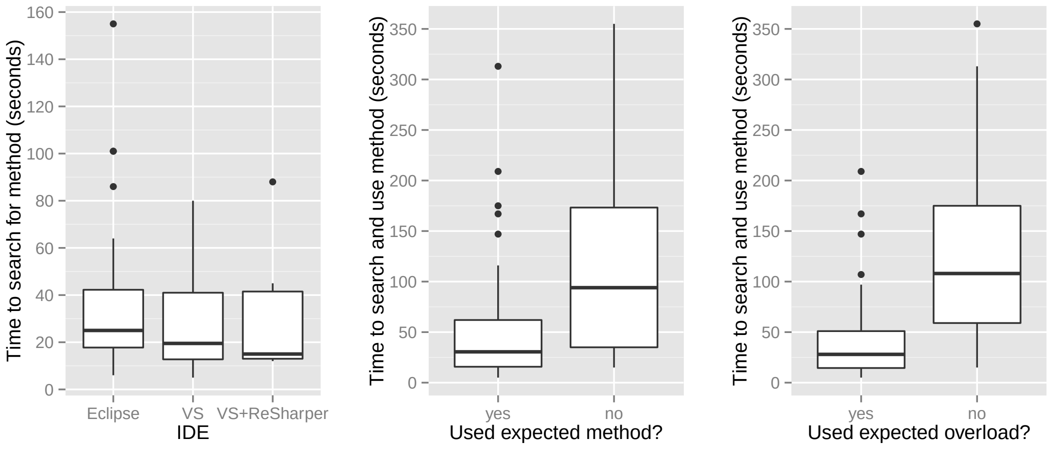


Figure 4.8: Boxplots for searching and using methods: (a) comparison of IDEs when searching *by list*, (b) optimal method used vs. non-optimal method used, (2) optimal overload used vs. non-optimal overload used

Influence of the IDE's Code Completion Mechanism – Performance

In addition to the number of optimal methods and overloads used, we compared the performance of the three variants. The results are shown in figure 4.8(a). The distribution of the values is very similar for all 3 IDEs, with the median time for searching a method *by list* being around 20 seconds, and there are no significant differences between any of the IDEs (p>0.3 for all combinations). Therefore it can be said that all code completion mechanisms are equal in performance. This is especially interesting since it shows that despite of the fact that the list in the Eclipse code completion window (showing 170 items for members of the *ZipFile* class, because all overloads are shown) is much larger than in Visual Studio (showing only 70 items, because overloads are not shown), users are not significantly slower in finding what they are looking for. When asked if the large number of methods and overloads bothered them, some of the Eclipse programmers answered that it didn't because when searching through the list, their eyes were just jumping over entries with the same starting signature. The fact that there is no difference in performance indicates that programmers who are experienced with a certain IDE have acquired such a selective way of reading.

It is important to not conclude from these performance results that in the end the code completion mechanism doesn't have any impact on performance. Since there were only relatively few cases (between 5% and 15%) where the optimal method or overload was not found, this is not noticeable in the overall search time statistics. To prove that using a non-optimal method or overload is significantly slower than using the optimal one, we can compare these two cases directly: Figure 4.8(b) shows a comparison of the search and usage times for methods, with a median time of 29 seconds for using the optimal method, compared to a median time of 94 seconds for using a non-optimal method. Figure 4.8(c) compares the usage of optimal and non-optimal overloads, with a similar result. There are significant differences in both cases

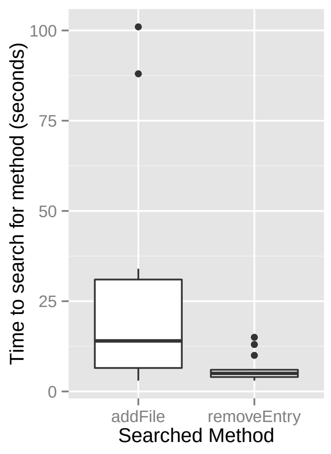


Figure 4.9: Boxplot: Time to search for method addFile vs. removeEntry (many vs. few overloads and methods with the same prefix)

(Wilcoxon: p=0.0109 for methods, p=0.0001 for overloads). This shows that both using a non-optimal method and using a non-optimal overload has a significant impact on performance, with the search and usage times being 2 to 4 times higher.

One may argue that the APIs were especially prepared so that non-optimal methods and overloads were significantly more difficult to use. But this was not the case – all methods were carefully chosen and designed following real world examples of ZIP APIs, and all of the methods are also useful in other specific use cases.

Influence of the Number of Methods with the Same Prefix

It has already been shown that, depending on the code completion mechanism, the number of methods with the same prefix has an influence on the chance of finding the optimal method/overload. In addition to that, we also want to analyze the influence of method with the same prefix independent of the specific code completion mechanism. We therefore compare task 1 (adding a single file to a zip file), with task 6 (removing a file from a zip file). While in both cases the prefix was equally easy to guess, there was a large number of methods with the prefix "add" (see figure 4.7), but only very few mit the prefix "remove". When searching for "remove", only 6 choices were visible, with the needed method at position 3 in Eclipse, and at position 1 in Visual Studio.

As shown in figure 4.9, programmers needed a median time of 14 seconds for searching the method `addFile`, compared to 5 seconds for `removeEntry`. They were significantly slower when searching the method `addFile` (Wilcoxon: n_1=15, n_2=17, p=0.001). Also, in the former case only 60% of the programmers found and used the optimal method and overload, while in the latter case all programmers used the optimal method and overload.

Table 4.8: Comparison of methods with different parameter numbers

Params	Calls	Median Time	Methods
1	48	00:07	AddFile, ExtractAll, ...
2	9	00:12	AddEntry, UpdateEntry (API 1)
4	2	00:28	UpdateEntry (API 2)

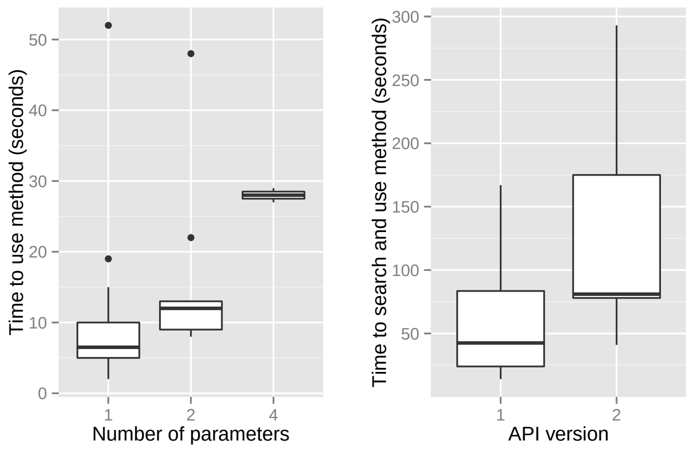


Figure 4.10: Boxplots: (a) comparison of usages times for methods with different numbers of parameters, (b) search and usage times for the updateEntry method, comparing API 1 (more overloads, less parameters) and 2 (less overloads, more parameters)

4.2.5 Using Methods

Concerning the usage of methods, we analyzed two different aspects. The first aspect is the impact of the number of method parameters on performance. Our study included the use of methods with 1, 2 and 4 parameters. An overview of these methods is given in table 4.8, showing for each number of parameters the number of observed calls, median time for method usage, as well as the method names. Figure 4.10(a) shows the method usage times in a boxplot. Unfortunately, for 4 parameters there were only 2 observations because programmers tended to not use the expected method. Also, tasks 4 and 8 were excluded from this evaluation because programmers didn't use the expected methods in most cases. Nevertheless, the observed data clearly shows that calling methods with more parameters also takes more time. When comparing 1 and 2 parameters, the times for 2 parameters are significantly higher (Wilcoxon: n_1=48, n_2=9, p=0.001).

The second aspect we analyzed is whether it is better to have a higher number of overloads or a higher number of method parameters. By that we wanted to find out if it is better to define many "default overloads" so that programmers only need to define few parameters in different use cases but may need to scroll through many overloads, or if it is easier when there are only few overloads but some additional parameters must be defined. During task 5 (updating the content of a file within an existing zip file) programmers had to use the `updateEntry` method. In API 1, this method had 8 overloads, and the optimal overload had 2 parameters. Parameters were a string for the entry name, and a string for the new content. In API 2, the method had only 4 overloads, but the optimal overload had 4 parameters. Parameters were the same as in API 1, plus two additional parameters to define which string encoding should be used and if the existing content should be overwritten or appended. These parameters would make sense for other tasks, but were unnecessary for the task at hand.

The surprising result was that, while with API 1 all programmers used the optimal overload, only half of them used it with API 2. The main reasons for this were that programmers expected that there would be a simple overload for this task and so they didn't even look at the seemingly complex 4-parameter overload, and that they found it irritating to need to define an encoding, and for that reason alone preferred to use a different overload. Other overloads needed a byte array or stream instead of allowing to hand over a string as content, so programmers had the additional effort of converting the string into one of these data types.

The results, as displayed in Figure 4.10(b), clearly show that a higher number of overloads (API 1) is better than a higher number of parameters (API 2). The median times of searching and using the method were 42.5 seconds for API 1 and 81 seconds for API 2. There is a high probability that the difference is significant (Wilcoxon: n_1=6, n_2=5, p=0.089).

Methods vs Properties

We wanted to show that using a property in .Net is equivalent to calling a method with a single parameter. This would mean that both concepts could be treated equally in the API Concepts Framework. Therefore, in task 7 API 1 contained a `setPassword` method to set the password, while API 2 contained a `Password` property. Unfortunately the study didn't produce a sufficient number of valid results for a statistical evaluation. But our observations during the study indicate that there is indeed no difference – in both cases programmers were able to find the method/property fast when searching for members in the `ZipFile` class, because both showed up immediately in the code completion window when typing "password". Further, the usage of both variants was also very straightforward, and no programmers encountered any problems there.

4.2.6 Impact of Programmer Experience

We analyzed for several variables whether they were influenced by the experience of the programmer. If experienced programmers were significantly faster than inexperienced ones, this would need to be taken into account when measuring API usability. It would also be a potential threat to the validity of the other results of this study.

Impact on Task Time

First we analyzed the impact of programmer experience on the overall task times. Figure 4.11 therefore compares the times to solve a task with the programmers' years of programming experience for all 20 study participants. Unexpectedly, it shows that there is no tendency of an

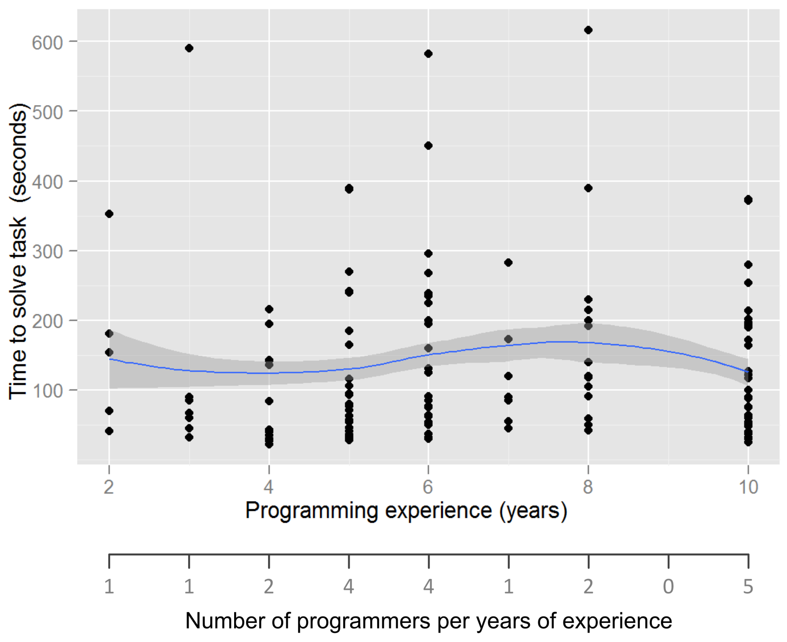


Figure 4.11: Influence of experience: Time to solve task vs years of programming experience

increased performance for more experienced programmers – the performance is about the same for all programmers from 2 to 10 years of experience.

We analyzed the data using *Pearson's product-moment correlation coefficient*, which is a measure of the correlation between two variables (a value of 1 means a total positive correlation, -1 a total negative correlation, 0 means no correlation). Additionally we evaluated the probability of the correlation by applying a *t-test*. The correlation value for programming experience and overall task time is -0.09 (p=0.143), which shows that there is no significant positive or negative correlation between the two.

The second aspect we analyzed was the programmers' domain knowledge. Programmers that already have experience in the area of the presented API (files, zipping, streaming) may have advantages because the needed concepts are familiar to them. For that we asked each programmer to state his/her self-assessed level of domain knowledge for the given problem domain, at a scale from 1 (lowest) to 5 (highest), according to the following rating:

1. no experience with files/streaming and zip APIs
2. some experience with files/streaming
3. familiar with files/streaming, may have tried a zip API
4. highly familiar with files/streaming, tried a zip API at least once
5. either frequently or recently been using one or more other zip APIs (different from the one presented)

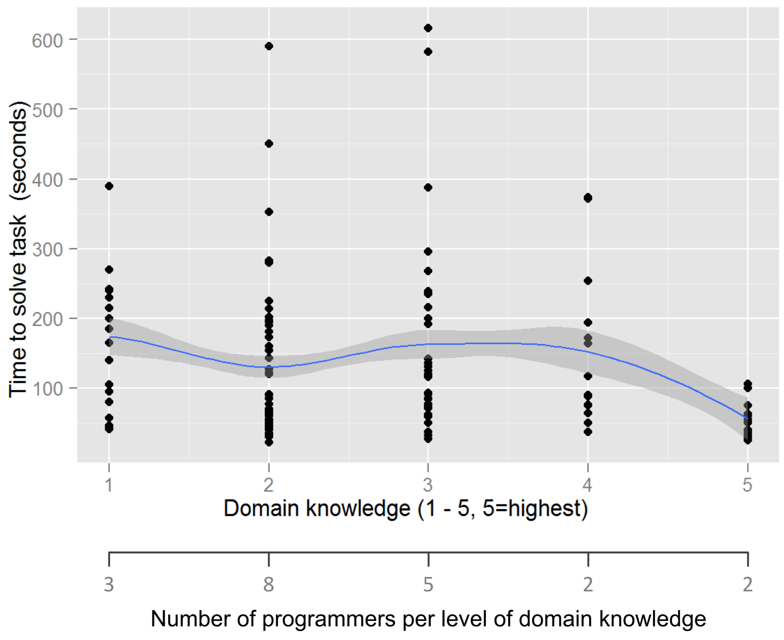


Figure 4.12: Influence of experience: Time to solve task vs programmer's domain knowledge

A majority of the programmers assessed their level of domain knowledge with the values 2 or 3. Only 2 of the programmers had been frequently using a zip API before.

Figure 4.12 shows the overall task times per level of domain knowledge. Again, most values are on a very similar performance level. A clear performance gap can only be seen at the highest level. The Pearson correlation coefficient for domain knowledge and overall task time is -0.15 (p=0.031), which shows that there is a very weak negative correlation between the two variables. When comparing the values of different levels of domain knowledge directly, a statistically significant difference can only be found with level 5. E.g. a comparison of level 2 and 5 shows that programmers with a domain knowledge of 5 were significantly faster (Wilcoxon: n_1=53, n_2=14, p=0.005). This makes sense, because someone that has frequently been using an API with very similar concepts, can be expected to benefit from a cross-API learning effect.

This shows that, surprisingly, except for programmers that are highly familiar with similar APIs, neither the programming experience nor the domain knowledge lead to any significant differences in performance. The programmer with the overall best performance results also was not the most experienced one; on the contrary, he only had 4 years of programming experience and a domain knowledge of 2.

Impact on the Number of Optimal Methods/Overloads Found

We further checked if the programming experience had any impact on whether the optimal method or overload was found and used. Figure 4.13 shows diagrams for the percentage of cases where programmers didn't find the optimal method or overload, depending on their years

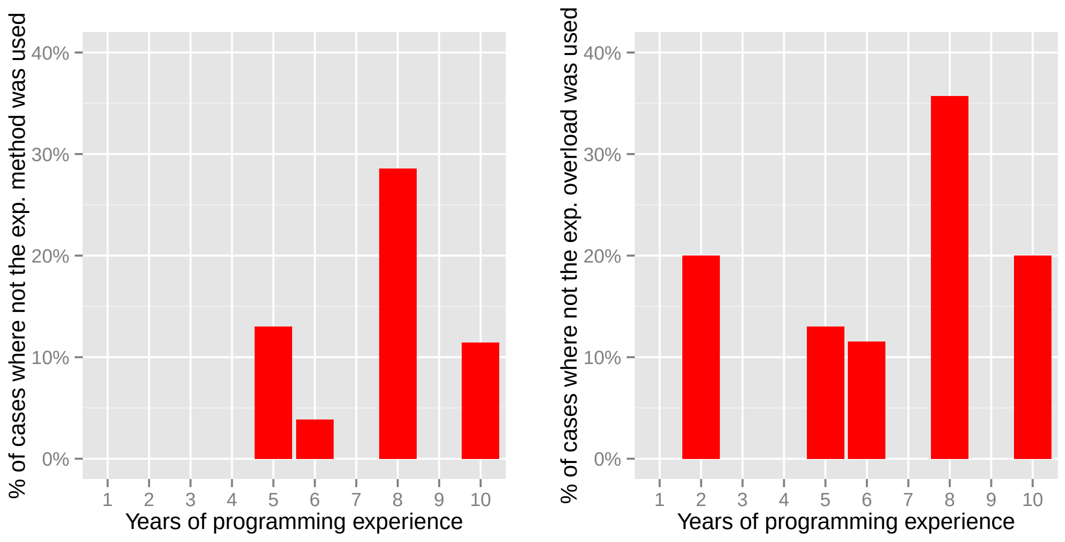


Figure 4.13: Percentage of cases where not the optimal method(1) / overload(2) was used, depending on the years of programming experience.

of experience. It shows that more experienced programmers were not better in finding the optimal method/overload.

Impact on Success Rate

We also reviewed if either the programming experience or the domain knowledge had any impact on the number of successful tasks. Out of the 20 programmers, 7 finished all 8 tasks successfully, 7 finished 7 tasks, 3 finished 6 tasks, and 3 were able to finish 5 tasks. Figure 4.14 compares this to the programmers' experience and domain knowledge. In both cases a tendency can be seen that more experienced programmers were able to finish a higher number of tasks. We added a small jitter to the y axis of the plots so that the dots would become better visible where multiple programmers finished with the same number of successful tasks.

The Pearson correlation coefficient is 0.43 (p=0.030) for programming experience and number of successful tasks, and 0.38 (p=0.049) for domain knowledge and number of successful tasks. So, in both cases a positive correlation between the variables can be found. This shows that, although more experienced programmers may not be able to solve tasks faster, their more wide spread knowledge helps them to better adapt to unforeseen situations and solve more tasks successfully. We also observed during the study that experienced programmers tended to have more confidence that they could solve a problem when it occurred, while inexperienced ones sometimes seemed insecure or helpless in such a case.

4.2.7 Programmer Satisfaction

After the test we asked each programmer to rate 3 aspects of the API on a differential scale, using the questionnaire described in section 4.1.2. The measured aspects were simplicity (simple ...

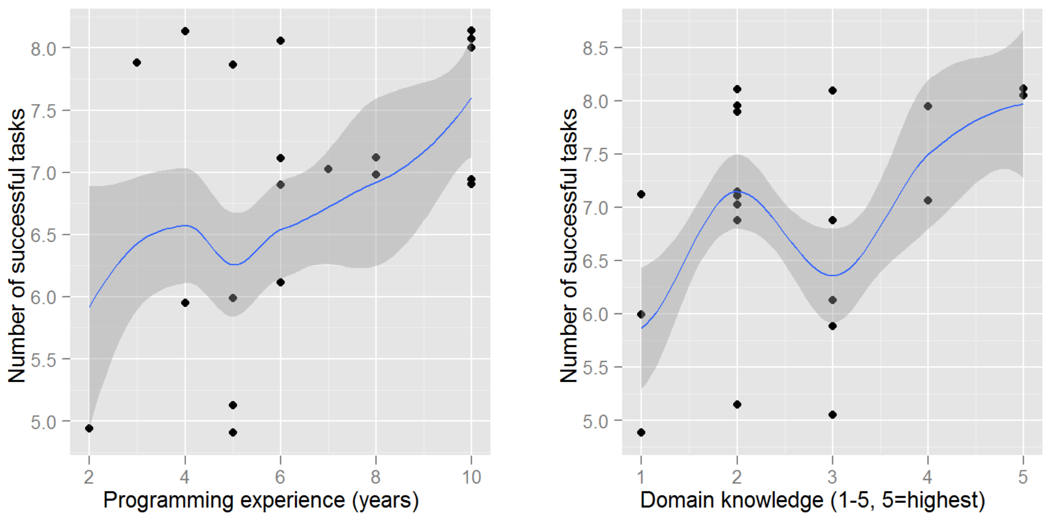


Figure 4.14: Influence of experience on the number of successful tasks: (a) successful tasks vs programming experience, (b) successful tasks vs domain knowledge

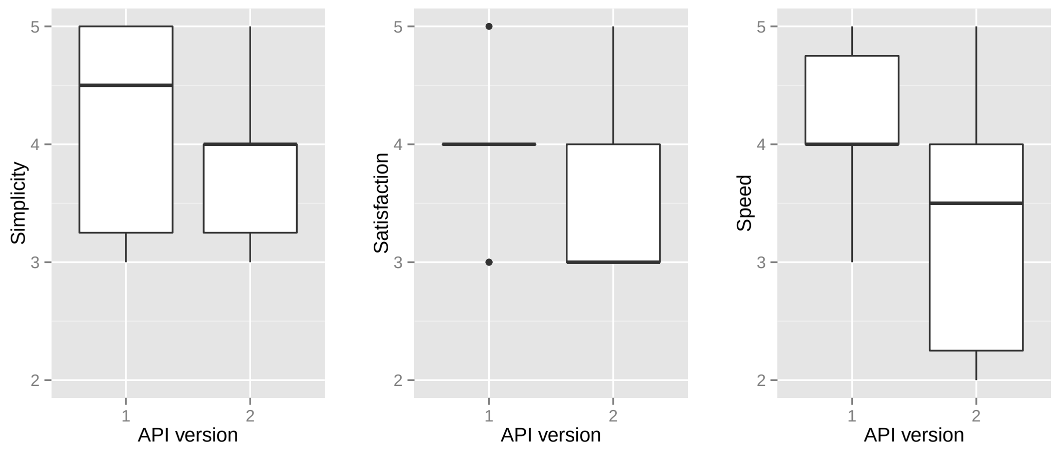


Figure 4.15: Boxplots showing the results of the questionnaire in study 1: (a) simplicity, (b) satisfaction, (c) speed

complicated), satisfaction (pleasing ... irritating/frustrating) and speed (fast to use ... slow to use). We converted this scale into a rating with 5 points for the positive, to 1 point for the negative term (meaning more points is better). The results for each aspect are shown in figure 4.15. Programmers gave a better rating for API 1 than for API 2. An analysis of all ratings combined shows that the results for API 1 are significantly higher (Wilcoxon: n_1=30, n_2=30, p=0.013).

This is a rather unexpected result, since API 2 was designed to be the simpler one, with fewer classes and methods. But the different types of instantiation seem to have left a much stronger impression, where programmers with API 1 (using a constructor) were significantly faster than with API 2 (using a static factory method). Especially the missing public constructor was often a cause for frustration with API 2, as described in section 4.2.2. In general, when something that the programmers expected was missing (like a public constructor, or a simple `updateEntry` overload), it had a much stronger negative impact than when there were elements that the users just didn't need.

4.2.8 Other observations

In addition to these results, we observed several other interesting details about usability-relevant API aspects:

Naming

During the tests we often observed that the naming of classes and methods was extremely important. If a class or method name didn't meet the programmer's expectations, he/she would need much more time finding it. An example for that is the method `save`, which was e.g. needed in task 1, as shown in Listing 4.1. 3 out of 20 programmers had problems finding this method because they were searching for a method with the name "write" or "close". Normally, it would take the programmers only a few seconds to find the method, but in these cases it took them 1 minute and more, meaning the search time was more than 10 times higher. Another example is the static method `read` which was used to open a zip file with API variant 2. 3 out of 10 programmers were looking for "open" instead, and took much longer finding the correct method although the list of static methods was very small.

Problems can also arise when the naming of a class member doesn't fit the member's type. We observed that when something is named like a command (e.g. "doSomething"), programmers tend to think that it is a method even though it may be a (possibly baldy named) field or property. In our study, the `ZipFile` class had a property named `ExtractExistingFile`, which allowed to specify whether an existing file should be overwritten when the `Extract` method was called. But because of its command-like name, some programmers tried to use it as a method, and did not recognize immediately that it was actually a property. We conclude from this, that whether the correct part of speech is used (e.g. subject or verb), has a significant influence on usability – while methods should be verbs (or commands in general), properties and fields should be subjects.

Another interesting case where problems occurred due to naming was task 8 (zipping a stream in memory). In addition to the `ZipFile` class, both API variants contained the classes `ZipInputStream` and `ZipOutputStream`. Although the optimal solution would have been possible with the `ZipFile` class, a vast majority of the programmers (17 out of 20) tried to solve the task with the stream classes. When later asked why they chose these classes, they stated that it mainly was because of the class names – the names of the stream classes sounded most fitting for the task.

Level of Abstraction

Although it was possible to solve the streaming task with the `ZipInputStream` / `ZipOutputStream` classes, it was much more difficult than with the `ZipFile` class, and the programmers were very surprised about these difficulties. They often stated things like "this can't be right, this is much too complicated – I'm sure there is an easier solution". The fact that programmers were so surprised can be explained by the unexpectedly low level of abstraction: With the streams they had to work with low-level byte arrays, while with the `ZipFile` class they could just conveniently hand over the file names without thinking about the file content. The level of abstraction is also mentioned in [39] as an important usability aspect, an we have discussed its measurability in chapter 3.

Unintuitive Ordering

During task 7, programmers had to zip a file with a password. To do that, programmers had to call the methods `setPassword`, `addFile` and `save` in the given order (see listing 4.1). If `addFile` was called before `setPassword`, the password was not set for this file. This is because zip files can contain both password protected and non password protected files (or files that are protected with different passwords). Out of the 20 programmers, 9 called `addFile` before `setPassword` when they first executed the code. Most of them were confused and did not understand why their code didn't work, because there was no indication that there was a required order for these two methods. Only 4 of the 9 programmers were able to find the error, either by reading the documentation of the `setPassword` method or by just trying around, but even if they found it, it cost them several minutes: While programmers that didn't encounter this problem finished the task in a median time of 80 seconds, programmers that had the problem needed 240 seconds. Whether a programmer got the order right at first try seemed to be pure coincidence in most cases. Only a single programmer stated afterwards that the order was completely clear to him from the start because he knew that zip files could contain both password-protected and password-free files.

Unexpected Solution Approach

Another interesting observation was made during task 5, which had the goal of updating an entry. This task could easily be solved by using the `updateEntry` method in the `ZipFile` class. Since there are similar methods e.g. for adding entries, it seems logical to be able to update entries in a zip file like this. But many programmers (11 out of 20) tried a different approach: Since an entry should be updated, they searched an update method first in the `ZipEntry` class, and only returned to the `ZipFile` class when they couldn't find anything suitable there. This may of course have been influenced by the task order (programmers learned about the `ZipEntry` class in the task directly before this one), but we expect it to also strongly depend in the goal that users have in mind:

- "add a file to the zip file" $\rightarrow$ `ZipFile` class
- "update a zip entry" $\rightarrow$ `ZipEntry` class

This again shows the importance of the context of use (user, task) for usability.

4.3 Interpretation of Results

This section summarizes which properties of classes and methods have an impact on usability, based on the results of the study. For every identified property, we give a short explanation why it is important and present suggestions what can be done to improve usability. Additionally we present ideas how the properties could be measured automatically.

4.3.1 Number of Classes in a Package

A high number of classes has a negative impact on performance when searching for a class and gives a negative impression to the programmers. When the number of classes is not much larger than what fits into the code completion window, programmers can easier get a quick overview of the classes available. Of course, it may not always be possible to keep the class numbers in a package this small because of classes that logically belong together and are best presented to the programmer in the same package. A maximum of 10 to 15 classes per package can be seen as optimum. In our study, search times nearly doubled for a three times higher number of classes. Though this may have been influenced by the explorative learning style that was used in this study.

Suggestions: The study showed that a simple restructuring into multiple (sub-)packages can help improving search performance and programmer experience. When only the main classes are left in the main package, programmers can easily identify what is most important in the API.

Measurement: The number of classes in a package can easily be checked automatically by analyzing the class structure in the API library.

4.3.2 Constructors vs Static Factory Methods

Constructors are easier to use than factory methods. This is true both when having the static methods in a separate factory class (as shown in [58]), as well as when having them in the instantiated class itself (as shown in our study). Most programmers also prefer a constructor over a static method. Depending on the expectations of the programmer, both variants can lead to problems, if the variant expected by the programmer is not used in the API. In our study, the most confusion was caused when programmers were not able to instantiate a class by constructor, because there were no clear compilation error messages and the private constructors showed up in the code completion window. We also showed that this is independent of the programming language. Although a factory may be better from the viewpoint of the API developer in terms of API evolvability [188], from the viewpoint of the user a constructor is the better choice.

Suggestions: Constructors should be preferred over static methods. A good solution may be to have both variants to fit different programming styles and expectations. To avoid confusion, a constructor should always be present where possible.

Measurement: Within a given implemented scenario, the usage of a constructor can easily be recognized by the `new` keyword. The usage of a factory method could be recognized when a static method is used that generates an instance from an API class.

4.3.3 Number of Class Members and Naming

Even a high number of class members only has a low impact on performance when searching a method. In our study, there were only marginal performance differences between API 1 with about 150 members and API 2 with about 50 members in the `ZipFile` class. This is because most of the time programmers are searching with a specific method name (or at least prefix) already in mind, making the overall number of members mostly irrelevant. The number of member becomes relevant in two cases: First, when a method cannot be found by guessing and therefore the programmer needs to go through the whole list, in which case the overall number of members directly affects the search time. Second, when there is a high number of members with the same prefix, making it harder for the developer to identify the correct method although the correct prefix has been guessed.

For example, programmers had difficulties when searching for the method *addFile* because there were many methods starting with the prefix “add”. Although they were searching with the correct prefix (which indicates that the method name is actually well chosen), programmers often could not find the method because there was a large number of other methods still visible in the code completion window. They had most problems when the desired method was so far down in the list that it was not directly visible without first scrolling down.

Suggestions: Much more than a low number of members, it is crucial that *well chosen* and *distinctive* member names are used. Non-Distinctive members counteract the searching by hint and should therefore be avoided. This also limits the overall number of methods in a class, since distinction is getting harder and harder the more methods a class contains. In the evaluated IDEs the code completion window contained about 10 to 15 elements, so when an important method appears further down than 10th position it should be considered to either rename the method, or to rename/remove other methods placed above, to improve usability.

Measurement: As already discussed earlier in this thesis, there is no simple way to check if method names are well chosen. But an aspect that can be measured in terms of name distinctiveness is the number of members with the same prefix, where a high number indicates possible usability problems. What is especially interesting is the number of elements that are ordered above the method in the code completion window, depending on the IDE. In Eclipse, this could be calculated by the overall number of members with the same prefix, in Visual Studio by the number of methods without counting the overloads.

4.3.4 Number of Overloads

Overloads were especially hard to find in cases where an overload was further down in the list, either because of an unexpectedly large number of parameters, or because of an unfavourable alphabetical ordering. Unfortunately there is no possibility for the API to influence the alphabetical ordering. What can be influenced by the API is the number of parameters by having the API define “default overloads” with fewer parameters, so programmers can find a simple overload on top of the list if they expect one.

Suggestions: Depending on the IDE, a large number of overloads can lead to usability problems which should be taken into account when creating APIs in the corresponding programming language:

With Visual Studio we observed that programmers tended to not scroll down further when they expected to find an easy to use overload, e.g. one with just a single parameter, while actually the best suitable overload had three parameters. Sometimes they did not notice that the method has overloads at all. It should be tried to keep the number of overloads per method low, because with more than 3 overloads, programmers tended to not even look through the whole list. Important overloads should be prevented from being far down in the list, e.g. by removing others placed further up that may be rarely used. Also, the .Net feature of *optional parameters* can help keeping the number of overloads to a minimum.

With Eclipse, due to showing methods and overloads in a single list, overloads can push other important methods out of view in the code completion window. Therefore, methods that are listed above important other methods with the same prefix should not define too many overloads.

Measurement: Very similar to the number of methods with the same prefix, it is especially interesting how many overloads are placed above in the code completion window, which can automatically be counted by ordering the overloads first by parameter count and then alphabetically.

4.3.5 Number of Method Parameters

It is clear that a higher number of method parameters also means that it takes longer to use a method. For parameter counts between 1 and 4 parameters, our study shows that every parameter takes about 5-8 seconds of the programmer's time. Parameters that were not expected by the programmer tend to take more time than expected ones, since they require additional thinking, and tend to make the programmer look for a better suitable method or overload.

Suggestions: The number of method parameters should be kept as low as possible. Overloads defining default values should be provided for parameters that often don't need to be defined explicitly. In our study, a high number of overloads proved to be significantly better than needing to provide a high number of parameters.

Measurement: The number of parameters of a method can easily be measured automatically.

4.3.6 Learning Effect

The learning effect has a high impact on the overall performance of a programmer. When a programmer has used a class/method once or twice already, he/she doesn't need to search for it any more, meaning the search time is reduced to zero. Also, the time to use the class/method is reduced by a certain degree. How much this time is reduced depends on the complexity of the class/method. While it will be likely that the programmer will be faster when using a method with 4 parameters for the second time, there won't be a big change for a method with 0 parameters. Since searching for a class/method takes up about half of the time, it can be expected that the learning effect always leads to at least a 50% time reduction. It can be expected that the maximum learning effect is reached with the third usage, and that there is no significant further performance improvement afterwards.

Suggestions: If an API contains only few different concepts (classes/methods), and the programmer often reuses these concepts, a high learning effect can be guaranteed.

Measurement: It can be automatically recognized if concepts are used multiple times within a given scenario. A high number of different concepts needed for a single scenario could be an indicator of bad usability. To be able to calculate the complexity for a programmer that has already used an API, it should be possible to define the number of previous usages as an input parameter for the method - this would allow the programmer to evaluate whether he/she should learn a new API, or should prefer to use one he/she already knows.

4.3.7 Programmer Experience

Neither the years of programming experience nor the domain knowledge have a significant impact on performance or on the chance of finding the optimal method or overload. Further, both only have a very weak impact on the number of successful tasks. This is a surprising result, also because there were actually large differences between some programmers, with the fastest requiring only 8 minutes for all tasks combined, and the slowest requiring 30 minutes. This seems to partly depend on the programming style – programmers that were giving more thought before starting to code were generally slower than ones that started coding directly. The test environment is of course also an influencing factor – some programmers were considerably more nervous than others, nervousness lead to lack of concentration, being less straightforward when programming, and often being unsure if something is right. While in most cases the nervousness went down significantly after the first or second task, in some cases (especially when a programmer was less experienced, or when something went wrong right at the start) it persisted through the whole test. Another factor just seems to be what can be called "talent" – in some rare cases the experimenter was simply amazed how fast a programmer was able to write down a working solution. But all these factors were well distributed among the test participants, so there is no danger of them having a negative influence on the statistical results about programmer experience.

Concerning the API Concepts Framework, this is a positive result, because it means that the experience of the programmer (except of course for the experience with the API in question) can be disregarded, without risking to loose a broad applicability of the method. This doesn't mean though that it is guaranteed that experience and domain knowledge have no impact at all, since the presented results strongly depend on the way they have been measured. The years of programming experience do not reflect how intensively a person has been programming (e.g. part-time or full-time), or what specific things have been learned in that time. The self-assessed domain knowledge value poses the danger that what a programmer thinks about him/herself may not be accurate. Nevertheless, the results at least prove that a high correlation is very unlikely.

4.3.8 Influence of Code Completion Mechanisms

The study results show that there are significant differences between the evaluated code completion mechanisms. When searching for methods, a splitting in two steps, where first the method names and then the method overloads are shown, is superior to presenting all information in a single list. When searching for overloads, the presentation in form of a list is better than when only a single overload can be viewed at once. Considering these results, the best of all evaluated code completion mechanisms can be found in ReSharper.

The results also show that if the way an API is present by the code completion mechanism is not taken into account, this can lead to usability problems. The discovered problems are that more often non-optimal methods or overloads are chosen. This results in more complex code, as programmers have more effort fulfilling the method requirements since a higher number of parameters must be provided, parameters are more complex, or even multiple other methods must be used instead. The resulting code gets harder to read and more error prone. Also, the performance results showed that using a non-optimal method or overload takes significantly more time than using an optimal one.

We discovered that considerable improvements could be made in the ordering of overloads. While all explored IDEs order overloads first by number of parameters and then in an alphabetical ordering (starting with the name of the first parameter's type), this doesn't seem intuitive for most programmers. Instead of an alphabetical ordering, items could for example be ordered by the simplicity of the parameter types, e.g. a string parameter would always be ordered above a complex type. This was something that some programmers expected during the study, and were surprised that simple parameters were not standing at the top of the list. Another interesting possibility would be to define the ordering of overloads (or methods as well) as meta information in the API itself. Also, existing suggestions as described in section 2.2.2 could be evaluated if they can be used for ordering overloads.

Since our goal is measuring and improving the usability of APIs, and not improving code completion mechanisms, we will not investigate this matter further. But of course it is an interesting aspect for future work.

4.4 Study Conclusions

In the presented study we found several factors that influence that usability of API classes and methods. For searching and instantiating a class, we showed that the number of classes in the same package, as well as the type of instantiation (by constructor or by static method) play a significant role for usability. For searching and calling a method, we showed that the number of other class members with the same prefix and the number of parameters are most relevant for usability. When using a class or method multiple times, we showed that the learning effect significantly increases performance. Further, the study results indicate that the experience of a programmer has no significant influence on performance, which is an important fact for the feasibility of the API Concepts Framework.

Overall, the study results give a good overview over the usability of low-level concepts. The results will be used in chapter 6 for deriving concrete complexity values for LCs and measurable properties.

CHAPTER 5

Usability Study 2: Configuration-Based APIs

After gaining a better understanding about usability aspects of basic API concepts (low-level concepts, see section 3.2.1) in the first study, the goal of the second study is to analyze the usability of concepts with higher complexity levels (high-level concepts, see section 3.2.2). We are interested not only in finding out more about the usability of each concept, but also in comparing concepts with each other, since many of them present alternatives for solving similar problems. What is important for an objective comparison is that a use case is found where all concepts that should be compared are equally suitable. And of course, we want to compare concepts that are used in a large number of APIs.

A review of existing APIs and literature comparing APIs show that there are many areas where APIs are used for some kind of configuration. Examples are APIs for object-relational mapping like Hibernate[1], APIs for dependency injection like Unity[2], or APIs for communication like the Windows Communication Foundation[3] (see Table 5.1). A large-scale analysis of open-source Java projects presented in [121] lists five APIs under the top ten most used APIs: three for XML serialization, two for logging and one for unit testing.

When looking at such "configuration-based APIs", many of them share the same basic design concepts, independent from the area of usage. We identified three design concepts that are used for such APIs: The first is *XML*, where the whole configuration is not written in code, but stored in a separate XML file. The second is *annotations* (or *attributes* in .Net), where the configuration is done by annotating code elements with additional information. The third is *fluent interface* [71], which tries to make use of the natural language by defining methods that are concatenated to form a readable sentence. There is not yet any research concerning the usability of any of these concepts in the context of APIs. Our goal for the second study therefore is

[1]http://www.hibernate.org
[2]http://unity.codeplex.com
[3]http://msdn.microsoft.com/en-us/library/dd456779.aspx

Table 5.1: Examples for configuration-based APIs

API for	Configures	Then does
Dependency Injection	bindings from interfaces to classes, what is injected during instantiation	create instances
Serialization	which fields are serialized	serialize/deserialize objects
Object-Relational Mapping	mappings from code to db	insert, update and read data to/from the db
Communication	which messages are sent, which transport layer is used	send/receive messages
Logging	the log level, where the log is written, the message format	write messages to the log
Unit Testing	the conditions for a successful unit test	execute tests and check if the conditions are met
Mocking	the behavior of a mock object for unit testing	act with the configured behavior, verify if the mock object was used as expected

to evaluate the usability of the three API concepts *XML*, *annotations* and *fluent interface*, and compare them to each other.

Like in the first study, we are using an *A/B Test*, only this time with three different API variants, where each variant implements one of the three concepts. Also, we again use a *between-subjects design* to prevent cross-API learning effects, apply the *thinking aloud* method which helped greatly in understanding the users' behaviour, and let the users fill out a *post-test questionnaire* very similar to the one used in the first study.

We want to show in this study which concept performs better or worse in which situations, and which factors influence usability for each concept. One big difference to the first study is that there are hardly any known measurable properties for the concepts in question. It is all the more important that the design of the study is done very carefully so that yet unknown measurable properties will come to light. Table 5.2 defines which aspects are to be evaluated in this study. Since one of the goals defined in section 3.3.1 was to also conduct a study where users are using a documentation for learning, we integrate it in this study.

The following sections present the design and execution of the study, as well as a statistical evaluation and interpretation of the results including advantages, disadvantages and measurable properties for each concept. The results of this study were published in [170].

Table 5.2: Properties and Aspects to be Evaluated in the Usability Study

Property / Aspect	**Way to Measure**
How do the **three concepts XML, annotations and fluent interface compare to each other**?	Design three different API variants for use cases where each concept presents an equally fitting solution.
How usable is each concept in **different situations**?	Define multiple tasks, ranging from simple to complex ones.
How do users use a **tutorial/documentation** when learning an API? What are the differences to learning by exploration?	Give the user access to a tutorial/documentation. Use eye tracking to analyze in more detail how users read API documentation. Compare the user behavior to the results of study 1, where no documentation was available.
Are **annotations comparable to other concepts**: instantiation and field access?	Include tasks where annotations with one or more fields are used. Compare the times to other instantiation times, maybe also to times from study 1.
Are **XML elements/attributes comparable to other concepts**: instantiation, method call and field access?	Design the XML API so that elements and attributes map to methods and method parameters in the fluent interface API, and to classes and fields in the annotations API.
What **properties of a method chain** in the fluent interface have an influence on usability?	Define method chains with different lengths and number of method chaining options.
How strong are the **learning effects** for each concept? Are there differences in learning effects depending on the concept?	Present tasks in a meaningful order (i.e. from simple to complex, but with repeated use of the same concepts) to be able to monitor the learning effect.
Is there a **difference between Java and .Net**?	Design equal APIs for Java and .Net, have tests with both programming languages.

5.1 Design of the Study

5.1.1 Design of the APIs

From the areas shown in Table 5.1 we chose *dependency injection* [70] (further in short: DI) as a use case for building the APIs for this study. The reasons we chose DI are that it is widely known, tests are easy to setup because there are no external dependencies (e.g. for testing an object-relational mapping API, a database would be needed) and there are many different APIs available for all three concepts that need to be implemented. To identify which tasks are most often done with a DI API, as well as to see how the three concepts are used in this context, we looked at a number of existing APIs. We examined Spring[4], Java EE[5], Google Guice[6] and PicoContainer[7] on the Java side, as well as Unity[8], Ninject[9], Castle Windsor[10], StructureMap[11], AutoFac[12] and the Managed Extensibility Framework (MEF)[13] on the .Net side.

We chose the tasks that were most common with existing DI APIs and ordered them so that users can apply their knowledge from previous tasks, allowing us to monitor the learning effect. To be able to compare the different API variants with each other, and relate differences only to the different concepts with which the APIs have been built, it is very important to make everything else about the APIs as similar as possible. Therefore we evaluated the cognitive steps required for solving each task, and implemented the APIs following these steps as closely as possible. Further, we used the same terms wherever possible, so that there is no impact on the study results from such differences (as the first study showed, naming can have a strong impact if the expectations of the users are not met). We therefore closely followed the domain language dictated by existing APIs.

The following 6 tasks had to be solved in the given order. The API-specific codes are labeled with A for annotations, F for fluent interface and X for XML. The classes on which the DI task should be performed are labelled with SPEC (for "specification"). In all three APIs and in all tasks, first an instance of the *DIContainer* class had to be created. With the annotations API, it was additionally necessary to specify from which assembly/jar file the bindings should be loaded. With the XML API, the path to the XML configuration file needed to be specified. At the end of all tasks, the class for which the binding/injection was defined, needed to be instantiated. In the following code examples only the part of the code is shown that is specific to the particular task.

1. Creating a simple binding from an interface type to an implementation type. The cognitive steps for this task are (1) choosing the interface type for the binding, (2) choosing the

[4] http://projects.spring.io/spring-framework
[5] http://www.oracle.com/technetwork/java/javaee
[6] http://code.google.com/p/google-guice
[7] http://picocontainer.codehaus.org
[8] https://unity.codeplex.com
[9] http://www.ninject.org
[10] http://www.castleproject.org
[11] http://www.structuremap.net
[12] http://autofac.org
[13] http://msdn.microsoft.com/en-us/library/dd460648(v=vs.110).aspx

SPEC
```
public interface ILogger { ... }
public class ConsoleLogger : ILogger { ... }
```

A
```
[BindingFor(typeof(ILogger))]
public class ConsoleLogger...
```

F
```
container.Bind<ILogger>().To<ConsoleLogger>();
```

X
```
<DIContainer>
  <Bindings>
    <Binding type="Classes.ILogger"
        bindTo="Classes.ConsoleLogger"/>
  </Bindings>
</DIContainer>
```

Listing 5.1: Codes for task 1

SPEC
```
public class ConsoleLogger : ILogger { ... }
public class FileLogger : ILogger { ... }
```

A
```
[BindingFor(typeof(ILogger), Name="console")]
public class ConsoleLogger...
[BindingFor(typeof(ILogger), Name="file")]
public class FileLogger...
```

F
```
container.Bind<ILogger>().To<ConsoleLogger>().Named("console");
container.Bind<ILogger>().To<FileLogger>().Named("file");
```

X
```
<DIContainer>
  <Bindings>
    <Binding type="Classes.ILogger"
        bindTo="Classes.ConsoleLogger" Name="console"/>
    <Binding type="Classes.ILogger"
        bindTo="Classes.ConsoleLogger" Name="file"/>
  </Bindings>
</DIContainer>
```

Listing 5.2: Codes for task 2

implementation type (in the annotations API this is done implicitly by placing the annotation). In all variants the term “bind” or “binding” is used to convey the binding. Since only very few classes/methods/fields or XML elements/attributes are involved, this task is considered a simple task. See listing 5.1.

2. Creating named bindings. This task is very similar to the previous one, with the extension that two bindings instead of one need to be defined, and each binding needs to be assigned a name (which is one additional cognitive step). It aims to show how well the user is able to apply what he/she has already used, with only a minor extension to the previous task. See listing 5.2.

SPEC

```
public class Service {
  public Service() {...}
  public Service(ILogger logger) {...}
}
```

A

```
[Inject]
public Service(ILogger logger)...
```

F

```
container.WhenInstantiating<Service>()
         .UseConstructorWithTypes<ILogger>();
```

X

```
<DIContainer>
  <Instantiations>
    <Instantiation type="Classes.Service">
      <Constructor>
        <Param name ="logger" />
      </Constructor>
    </Instantiation>
  </Instantiations>
</DIContainer>
```

Listing 5.3: Codes for task 3 (binding is the same as in task 1)

3. Defining for a class which constructor is used (constructor injection), and injecting a single parameter. The cognitive steps for this task are (1) creating a binding for the type that should be injected (this step is equal to task 1) and (2) defining which constructor should be used. In the fluent and the XML API the terms “instantiation” and “constructor” are used for that. Further, in these APIs the constructor is defined by its list of parameters (although this makes the XML code quite complex, it seems to be the best way to do it, see e.g. the Unity framework for comparison). For the annotations API this is not necessary because the constructor is sufficiently defined just by placing the annotation. Therefore the single term “inject” is used, which is often found for this purpose in other DI frameworks. With the increased number of API elements that need to be used, this can be considered a complex task. See listing 5.3 (the code for creating the `ILogger` binding is not displayed, it is the same as in task 1).

4. Defining which constructor parameters are injected by using named bindings. This is similar to task 3, but imposes further complexity on the user by requiring additional API elements to be called. The cognitive steps are (1) defining two named bindings like in task 2, (2) defining the constructor that is used for instantiation, and (3) defining for each constructor parameter which binding is used for injection by specifying the binding name. The term “inject” is used to convey the constructor parameter injection, except for the XML API, where “binding” is used since a verb may be unintuitive for naming an attribute. See listing 5.4 (the code for creating the `ILogger` bindings is not displayed, it is the same as in task 2).

5. Creating a binding with a singleton scope (so only a single instance is created for multiple calls). The code is similar to task 1, with the addition that the users need to define the

SPEC

```
public class Service {
  public Service(ILogger consolgeLogger, ILogger fileLogger)
  {...}
}
```

A

```
public Service([Inject("console")] consolgeLogger,
               [Inject("file")] ILogger fileLogger)...
```

F

```
container.WhenInstantiating<Service>()
       .ForConstructorParam("consoleLogger").Inject("console");
       .ForConstructorParam("fileLogger").Inject("file");
```

X

```
<DIContainer>
  <Instantiations>
    <Instantiation type="Classes.Service">
      <Constructor>
        <Param name ="consoleLogger" binding="console"/>
        <Param name ="fileLogger" binding="file"/>
      </Constructor>
    </Instantiation>
  </Instantiations>
</DIContainer>
```

Listing 5.4: Codes for task 4 (bindings are the same as in task 2)

A

```
[BindingFor(typeof(ILogger), Scope = Scope.Singleton)]
public class ConsoleLogger...
```

F

```
container.Bind<ILogger>().To<ConsoleLogger>()
  .InSingletonScope();
```

X

```
<DIContainer>
  <Bindings>
    <Binding type="Classes.ILogger"
        bindTo="Classes.ConsoleLogger" scope="Singleton"/>
  </Bindings>
</DIContainer>
```

Listing 5.5: Codes for task 5 (specification is the same as in task 1)

scope of the binding. At this stage, users should know the API structure very well, so this task aims to show how well experienced users can work with the API when they have only minor things to look up. See listing 5.5 (the specification is the same as in task 1).

6. Applying all of the functions used in previous tasks on a more complex class structure, with three different bindings and two constructor injections. This task introduces no new concepts, and aims to show how well the user has already learned to solve complex tasks with the API, requiring everything he/she has learned up until now.

Concerning the XML-based API, we investigated a number of existing APIs with XML configuration and found that a majority of them do not directly provide an XML schema. Even if a schema is provided, it needs to be added manually to the IDE in a separate step from referencing the API (e.g. in Visual Studio this feature is rather hidden). We therefore decided that it would be more representative for existing APIs to not provide an XML schema for the study participants.

In addition to the API itself, a short tutorial document was created for each API. Again attention has been paid to make the three tutorials as similar as possible, to minimize influence of the documentation structure on the study. Therefore, the sections, the positioning of code examples, and even the texts (except e.g. API-specific keywords) were kept exactly the same. We also carefully prepared the tutorial content so that the users would find everything that is needed to solve the tasks, but not find exactly the code that was needed in a single example (so it was not possible to solve a task by just copying something from the tutorial without understanding what it does). The tutorials for all 3 APIs can be found in appendix B.

5.1.2 Participants and Environment

Our study included 27 programmers (9 per API), which were recruited in the same way as in study 1. 17 were Java programmers using Eclipse as IDE, 10 were .Net programmers using Visual Studio. The study involved both programmers with academic and industrial background, which had experience with a variety of different APIs. The participants were between 22 and 46 years old and had between 2 and 20 years of programming experience. All programmers were recorded using a screen capturing software, and a supervisor was sitting next to the programmer to explain each task. The tests were designed as unittests, so users could evaluate the code at any time (a successful test run marks the end of each task). The code templates were therefore very similar to study 1, see listing 4.2.

The biggest difference to the first study was that additionally an eye tracker was used to capture the participants' eye movements, allowing us to add gaze replays to the screen capture data for evaluating how users read API documentation. This required a very specific hardware setup, as can be seen in figure 5.1. The participant's seat is placed on the left hand side, the supervisor's on the right. The eye tracker model used is a *Tobii X50*, which was gratefully lent out from the Interactive Media Systems institute at TU Vienna. Since being an older model, this eye tracker needs a special server software (which interprets the raw data coming from the tracker) that requires significant processor power, and a also powerful graphics card installed (newer models have this integrated directly into the hardware). To prevent this from influencing the performance of the test computer, the eye tracker allows a dual-computer setup, where one computer has the eye tracker software installed (the computer on the right hand side in figure 5.1), and a second computer (laptop in the middle) acts as the test computer where the capturing software as well as everything else that is needed for the test (Visual Studio, Eclipse, ...) is installed. Both computers are connected with a direct network cable. The test computer itself uses a dual-screen setup, where the participant is working on the big screen, and the supervisor can see the live eye tracking visualization on the laptop screen. This allows the supervisor to recognize when tracking doesn't work, e.g. because the participant has left the range of the eye tracker.

One unfortunate detail of the eye tracking software was that the tracking was stopped when the user pressed the ESC button, which could not be disabled in the settings. Even when being

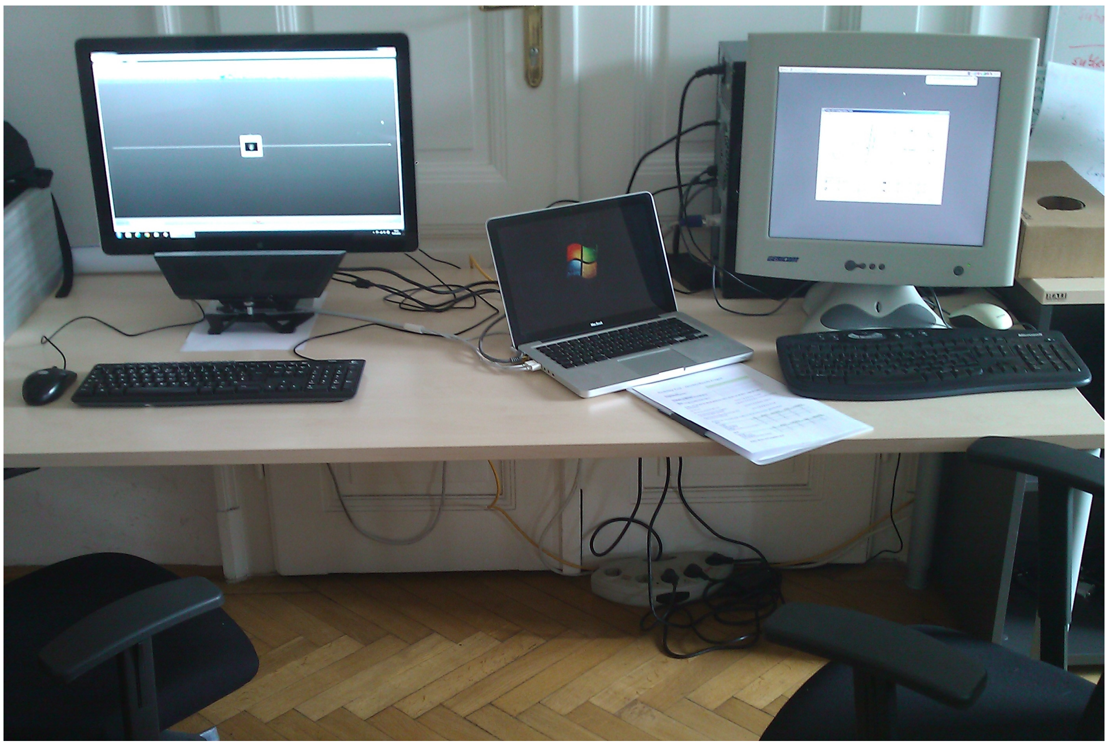


Figure 5.1: Test setup for study 2 including eye tracker

told not to press the button, users would just automatically press it when e.g. a window pops up in the IDE that they didn't want to see. To prevent this from disrupting the tests, a small paper hat was placed over the button (see figure 5.2), which, although shortly surprising users when they wanted to intuitively press ESC, successfully prevented any further disruption.

The sceen capture was again combined with a webcam video, as well as with the gaze replay data of the eye tracker, showing the user's "gaze points" (eye fixations) as red dots, as well as lines between the gaze points. An example screenshot is shown in figure 5.3. The webcam video was made half-transparent, so that no information is lost when users were looking at the lower right corner of the screen.

5.1.3 Measured Data

From the captured video data, we extracted times for specific steps that programmers had to do in each task, in the same way as we did in the first study (see section 4.1.5). These steps include instantiating the DI container, creating a certain binding, defining the constructor injection for a certain class, reading a certain part of the tutorial, running a test and searching for errors. For each of these steps, the time was measured, as well as the number of switches between tutorial and code, the number of switches between classes, and the place where the user searched for information (tutorial, code completion window, or none.)

When participants used the tutorial, we focused on identifying how long they took to un-

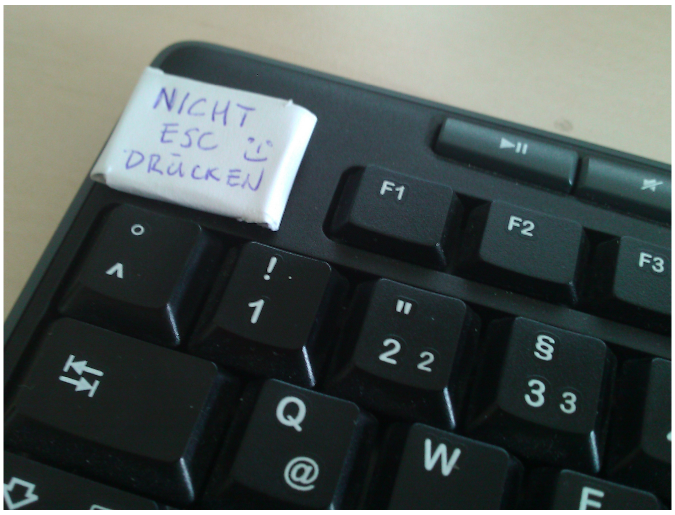
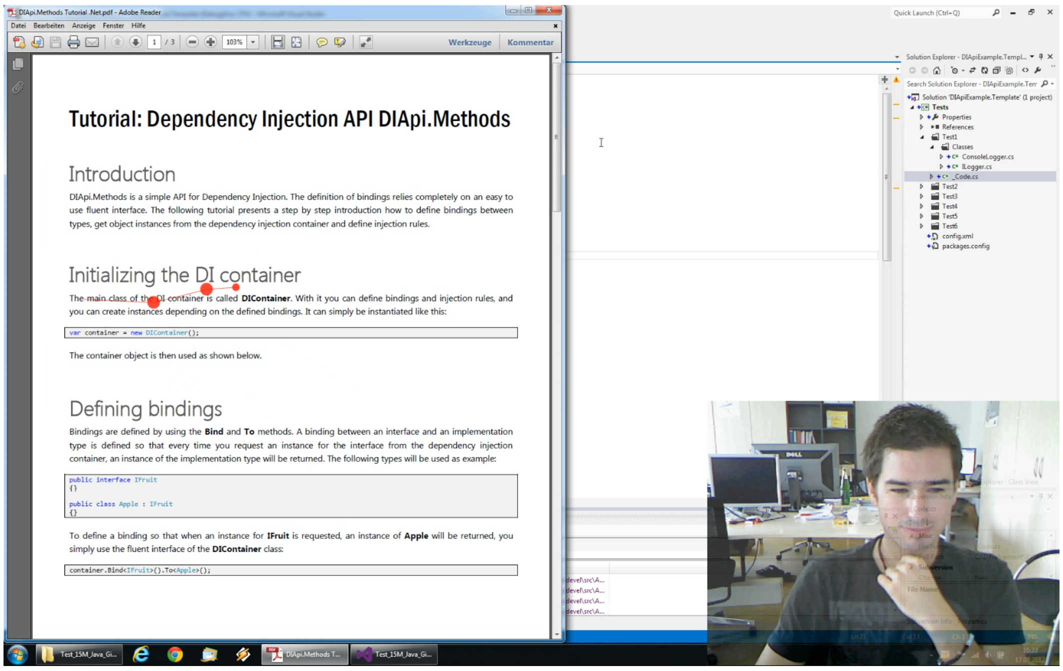


Figure 5.2: “Paper prototype” to prevent the user from pressing ESC


DIApi.Methods Tutorial .Net.pdf - Adobe Reader
Datei Bearbeiten Anzeige Fenster Hilfe
Werkzeuge
Kommentar

Tutorial: Dependency Injection API DIApi.Methods

Introduction

DIApi.Methods is a simple API for Dependency Injection. The definition of bindings relies completely on an easy to use fluent interface. The following tutorial presents a step by step introduction how to define bindings between types, get object instances from the dependency injection container and define injection rules.

Initializing the DI container

The main class of the DI container is called **DIContainer**. With it you can define bindings and injection rules, and you can create instances depending on the defined bindings. It can simply be instantiated like this:

```
var container = new DIContainer();
```

The container object is then used as shown below.

Defining bindings

Bindings are defined by using the **Bind** and **To** methods. A binding between an interface and an implementation type is defined so that every time you request an instance for the interface from the dependency injection container, an instance of the implementation type will be returned. The following types will be used as example:

```
public interface IFruit
{}

public class Apple : IFruit
{}
```

To define a binding so that when an instance for **IFruit** is requested, an instance of **Apple** will be returned, you simply use the fluent interface of the **DIContainer** class:

```
container.Bind<IFruit>().To<Apple>();
```

Figure 5.3: Example screenshot of the video evaluation from study 2

derstand certain parts of it, and how long they spent on reading either the running text or the included code examples. In this case the eye tracking data was especially helpful, because participants would sometimes put code and tutorial side-by-side on the screen, which would have made it impossible to detect what the user was looking at without this data.

At the end of the study, participants had to fill out a short questionnaire concerning their satisfaction with the used API, which was the same as in the first study (see section 4.1.2). Further, the questionnaire contained an inquiry about the participants' experience with other frameworks, where they could specify for a number of known frameworks how well they had used them already, on the following 5-point scale: never, read about it, tried it, used it rarely, used it regularly. The goal of this inquiry was to have a clearer estimation of the participants' domain knowledge. Further, they were asked which kind of API they have mostly used in the past (Annotations, Fluent Interface or XML), what they prefer, and what they think are advantages and disadvantages of each of the three design concepts. See appendix A for the complete questionnaire.

With 27 users and 6 tasks per user, 162 videos were evaluated. More than 4800 independent values overall were collected during the evaluation.

5.1.4 Study Execution

The steps that were gone through for each test run were mostly the same as in the first study (see section 4.1.4). Therefore, here only the steps are listed that were additionally necessary because of the integration of eye tracking (they do not interleave with the other steps). These steps roughly follow the suggestions given in [149].

1. **Preparation:**
    - Startup the eye tracker computer and connect it to the test computer.
    - Check if the eye tracker and monitor are correctly standing at their calibrated location by checking the markings that were attached to the table during the hardware setup.
    - Check if the eye tracker and the dual screen setup (supervisor screen showing the gaze data) are working correctly.
    - Make sure that the supervisor's screen is not in the user's line of vision (the constant movement on the screen could distract the user).
    - Prepare a stationary chair with no wheels or leaning or swivel capabilities, to prevent the user from moving around too much.
    - Prepare the ESC button safety hat.

2. **Introduction:**
    - Shortly inform the user about the eye tracking part of the study.
    - Ask the user if he/she has any eye problems, and what type of glasses he/she is wearing (if any). Especially bifocals, trifocals and layered lenses can be problematic for the eye tracker.

- Ask the user to try to not move around much during the whole session, because of the eye tracker's limited range.
- While calibrating the eye tracker, ask the user to follow the dots on the screen, and to try to sit in the same position that he/she wants to sit later during the study session. If it seems necessary, give the user a very short programming task, to bring him/her to sit in the right position.
- Warn the user that you might remind him/her to change the sitting position, or move his/her hand if it blocks the eye tracker.
- Explain the user why he/she should not press ESC.

3. **The test itself:**

   - Constantly monitor whether the gaze data is correctly recorded, or if the user may have moved out of the eye tracker's range. If that is the case, try to bring the user back into the correct position, by asking gently and without disrupting him/her too much.
   - Consider recalibrating the eye tracker and/or the user's position between tasks, when the data seems to be too imprecise, or when the user has deviated too much from his/her original sitting position.
   - Don't make the user uncomfortable when the quality of the eye tracking data is not satisfying. There may be users where the eye tracker simply has problems capturing the eye movements (e.g. because of glasses, as mentioned above).

5.2 Study Results

We evaluated all results with statistical data analysis methods. In contrast to study 1, the data showed no significant floor effect, and was normally distributed. Because of that, if not mentioned otherwise, a *t-test* was used, which is a parametric statistical hypothesis test for assessing whether one of two samples of independent observations tends to have larger values than the other.

In the following sections, the letters A, F and X are used as short forms for the three API variants (annotations, fluent interface, XML). The statistics and plots were again created using *R* and *ggplot2*, for the scripts see appendix C.

5.2.1 Performing Simple Tasks (Creating a Binding)

The action of creating a binding represents a simple task which can be solved using a single annotation, or only very few methods or xml elements. The basic code for this task is shown in listings 5.1 and 5.2. In the first task, users had to write exactly this code, in successive tasks they had to additionally define a binding name, with one additional value/method/xml attribute needed in the code.

Figure 5.4(a) compares the times for all tasks that were needed for first reading the tutorial section about bindings and then creating the binding. The median times are 42s for A, 34s for

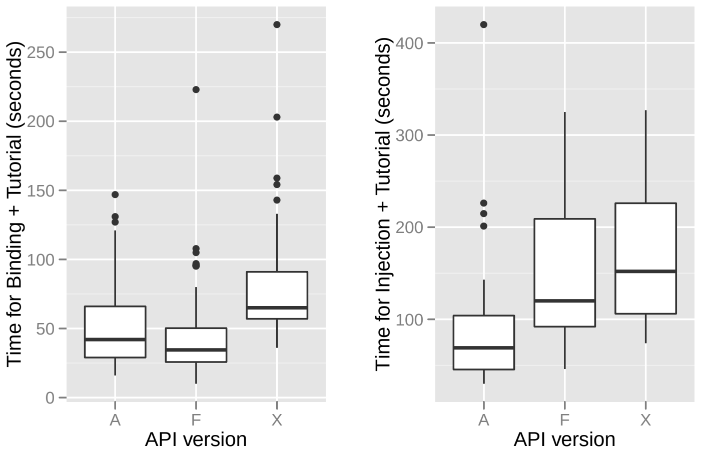


Figure 5.4: Boxplots: (a) Time needed for creating a binding incl. reading tutorial, (b) time needed for constructor injection incl. reading tutorial

F and 65s for X. A check for statistical significance shows that the times for both A and F are significantly smaller than X (p<0.001). When removing one extreme outlier of F, the times for F are also significantly smaller than A (p=0.042). So, for simple tasks users performed best with the fluent interface, closely followed by the annotations API. The large gap from X to the other two APIs is not unexpected, as users of the XML API spent much time writing the XML configuration without auto completion features available.

5.2.2 Performing Complex Tasks (Constructor Injection)

Performing constructor injection represents a more complex task. Users had to use multiple annotations, chain a higher number of methods in the fluent interface, and use more xml elements and attributes. The basic code for this task is shown in listings 5.3 and 5.4. Users first had to use constructor injection in task 3, and later in tasks 4 and 6. In tasks 4 and 6 users additionally had to specify which bindings are used for the injection of the constructor parameters, resulting in additional complexity.

Figure 5.4(b) compares the times for all tasks that were needed for reading the tutorial section about constructor injection and configuring the injection. The median times are 69s for A, 120s for F and 152s for X. A check for statistical significance shows that the times for A are significantly smaller than both F and X (p=0.017 for A<F and 0.002 for A<X). The times for F are not significantly smaller than X (p=0.195). The result of the annotations API being the best was expected because in this case much work is done implicitly just by choosing on which code element the annotation is placed. E.g. when the injection for constructor parameters needed to be defined, an annotation needed to be placed directly on the constructor parameter. With the other two APIs, users first needed to explicitly define the class and the constructor before they could even start with defining the injection of the parameters. This leads to the conclusion that the deeper within a hierarchy something needs to be configured, the greater the advantage of

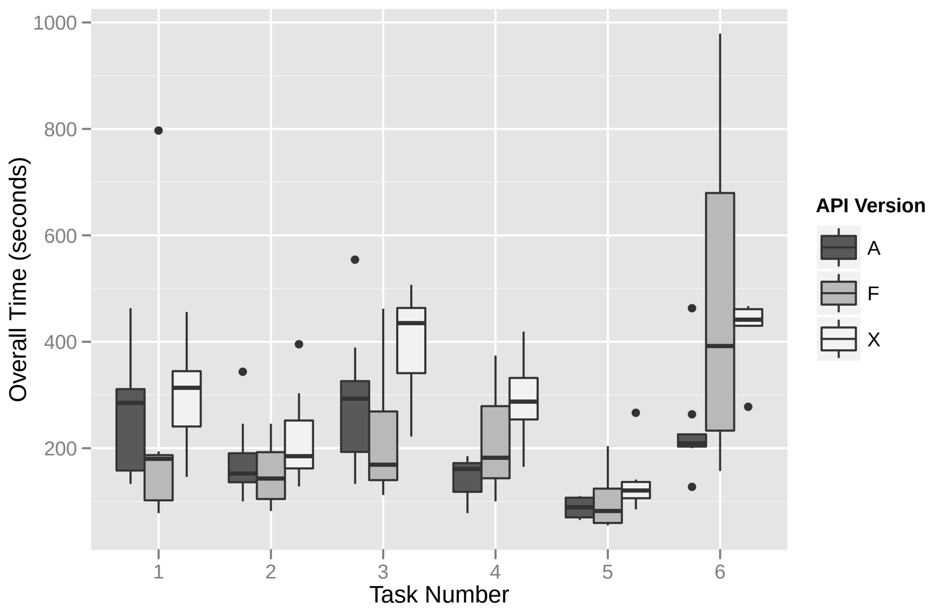


Figure 5.5: Boxplot: Overall times per task and API version

annotations against the other API variants will be.

What is surprising is that compared to simple tasks (see above) users with the fluent API were now much slower. This may indicate that a more complex method chain with a higher number of chaining options imposes a significant difficulty on the users.

5.2.3 Overall Times

Interesting details can also be found when looking at how much time users needed for each task depending on the used API. Figure 5.5 shows boxplots for the overall times per task, for each task comparing the three different APIs (see section 5.1.1 for a description of the tasks).

The first thing that can be noticed is the learning effect. This can best be observed when comparing tasks 1 and 2 (users needed to create a binding), as well as when comparing tasks 3 and 4 (users needed to use constructor injection). While both the annotations and the XML API show a clear speedup, there is hardly one to see for the fluent interface. This implies that the fluent interface has a more flat learning curve than the other two.

A second thing that confirms this suspicion are the times for task 6. For this task, users had to use everything they had previously learned, and we asked them to try to do it without looking into the tutorial if possible. Only for the XML API 5 of the 9 users had to look into the tutorial to solve the task, which was simply because they did not remember all needed XML elements and attributes any more, and were not able to use auto completion for this. When looking at the times for task 6, it can be noticed that while the spread is very small for A and X, it is very large for F. From this we interpret that users of A and X all had at this point gained a complete understanding of the APIs and knew exactly how to use them – so solving a task of this

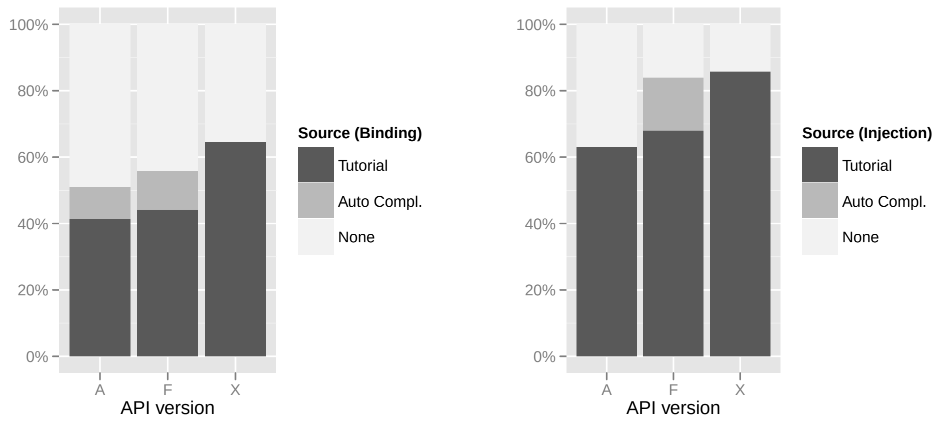


Figure 5.6: Which information source was used how often, compared by Task, for the action of (a) creating a binding, (b) constructor injection

complexity takes about 200-230 seconds with the annotations API, and about 420-470 seconds with the XML API. On the other hand, with the fluent interface some users were very fast (about 200 seconds), but others were extremely slow (up to 15 minutes), which means that many users still did not understand the API well enough to use it in a fast and efficient way. This could also be observed during the test, where users that had a good understanding of the API were very focused on the task and used API functions intuitively, while others needed to stop and think about how something is done with the API and also needed much more time searching for errors that were related to wrong API usage.

5.2.4 Information Sources

For each action the users needed to take during the test we analyzed where he/she acquired the information needed for that action. We distinguished between three different sources of information: The first was that users needed to look into the tutorial, the second that they used the auto completion mechanisms of the IDE, and the third was that they already had all the information they needed and therefore didn't need to look anywhere. For auto completion we expected that it would be most useful for the fluent interface, since it always presents the methods that can be chained next.

Figure 5.6 shows the evaluation of information sources for the actions of creating a binding (a) and constructor injection (b). Unsurprisingly the tutorial was used most often with the XML API (65% of the time for creating a binding, 85% for constructor injection), which is partly because of the missing auto completion. The users of the annotations API needed the tutorial the least often (40% and 60%), especially for the task of injection, where 20% more of the tasks were solved without using any information source than with the other APIs. Auto completion was most used with the fluent interface as expected (most clearly visible in figure 5.6(b)), but an advantage due to using auto completion can only be seen compared to XML.

Generally, the main advantage of using auto completion was actually not that users could

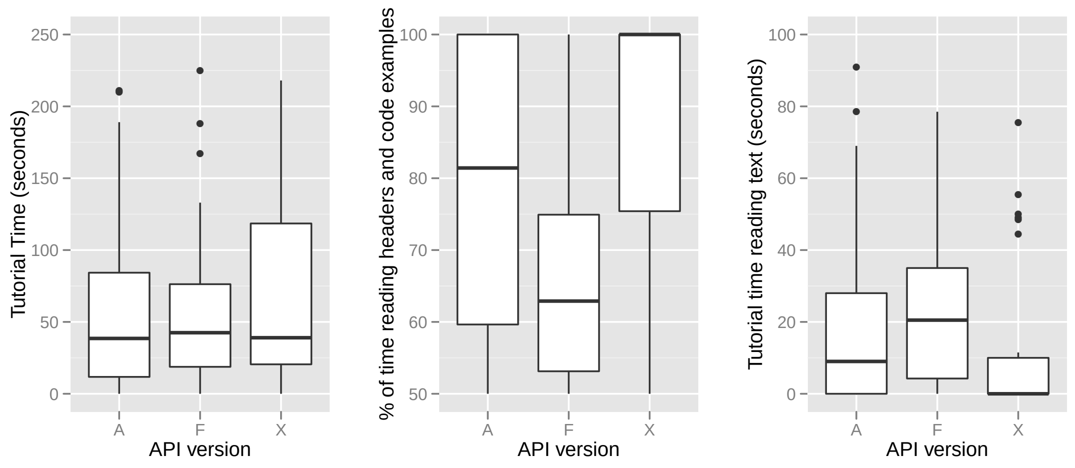


Figure 5.7: Boxplots: (a) Overall tutorial times per API, (b) Percentage of time spent reading only headers and code examples, (c) Time spent reading running text

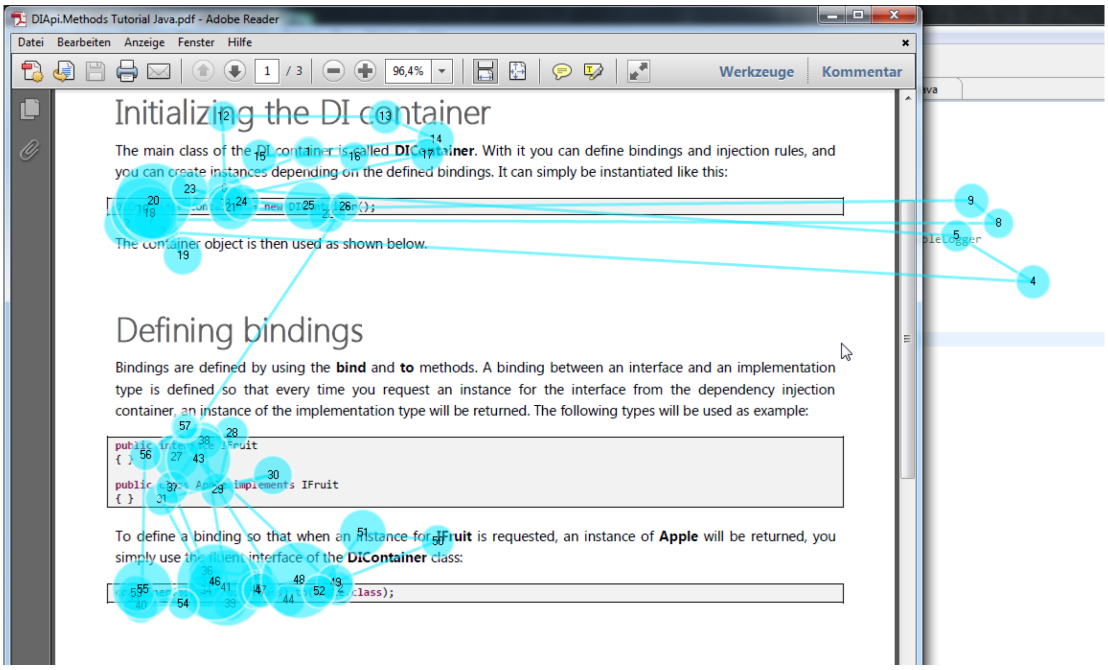
search for a method within the IDE, but simply that they didn't need to remember the whole class or method name. If the users only remembered the first few letters, they could still easily use a method by just auto-completing it. So, although it may appear in figure 5.6 that auto completion was rarely used, it was actually used very often, but only in 10-20% of the cases it was really used for searching. For simple completion of class, method and property names it was used in about 80% of all cases.

5.2.5 Time Spent in Tutorial

We analyzed how much time users spent in the tutorial and what they were reading there. The result can be found in figure 5.7(a). It shows that users spent about 30 to 35 seconds per task in the tutorial, with no significant differences between the three APIs.

The eye tracking data additionally allowed us to evaluate how much time users spent reading the running text, and how much reading only the headers and code examples (in all tutorials about 50% was running text, and the other 50% code examples). By doing that we intend to gain evidence on the self-explainability of each API. We observed that users almost always tended to look at the code examples first (after finding the correct section by scanning the headers), and only looked at the running text when there was something they didn't understand or the code behaved not as expected. Figure 5.8 shows a gazeplot that illustrates this behaviour. The blue dots mark eye fixations, in other words points that the user looked at, and the lines represent eye movements. Obviously most of the fixations are on code examples, while only very few are on the running text, and in this case only at bold words, or the words directly before bold ones.

Figure 5.7(b) shows the percentage of time that was spent in the tutorial only reading headers and code examples. While the median percentage is 81% for A and 63% for F, it is 100% for XML, meaning in over half of the cases users spent no time at all reading the text in the tutorial. This can also be seen when looking at figure 5.7(c), which shows the time spent reading



Initializing the DI container

The main class of the DI container is called **DIContainer**. With it you can define bindings and injection rules, and you can create instances depending on the defined bindings. It can simply be instantiated like this:

```
[illegible] = new DI[illegible]r();
```

The container object is then used as shown below.

Defining bindings

Bindings are defined by using the **bind** and **to** methods. A binding between an interface and an implementation type is defined so that every time you request an instance for the interface from the dependency injection container, an instance of the implementation type will be returned. The following types will be used as example:

```
public interface IFruit
{ }

public class Apple implements IFruit
{ }
```

To define a binding so that when an instance for **IFruit** is requested, an instance of **Apple** will be returned, you simply use the fluent interface of the **DIContainer** class:

```
[illegible]class);
```

Figure 5.8: Example gazeplot showing where a user looked in the tutorial.

running text. The median values are 9s for A, 21s for F and 0s for X. In this case we used the non-parametric *Wilcoxon Rank Sum Test* to check for statistical significance because the data shows a strong floor effect and is therefore not normally distributed. X is significantly lower than F (p=0.010), but not significantly lower than A (p=0.155), and A is very close to being significantly lower than F (p=0.052). This leads us to the conclusion that the XML code has the highest self-explainability because most of the time users didn't need to read any additional text to understand it. On the other hand, users of the fluent interface did not get enough understanding from just the code examples and therefore needed to read the whole tutorial more thoroughly.

The question may arise why the overall tutorial time for XML was not lower than the others. One reason is that due to the missing auto completion users often needed to switch into the tutorial just for looking how XML elements and attributes are written or how the exact structure looks like. Users of the XML API also switched into the tutorial much more often because of that.

5.2.6 Switches between Classes

We observed that the three APIs differed in the number of times that users needed to switch between different classes in the IDE. Figure 5.9(a/b) compares how often users switched between classes, for the tasks of creating a binding (a) and constructor injection (b). While A and X show only a slight difference between the two different tasks, there is a large one with F. This is because with the fluent interface, users only switched between classes when they wanted to take a look at the class contents. This was mostly when doing constructor injection, to look at the

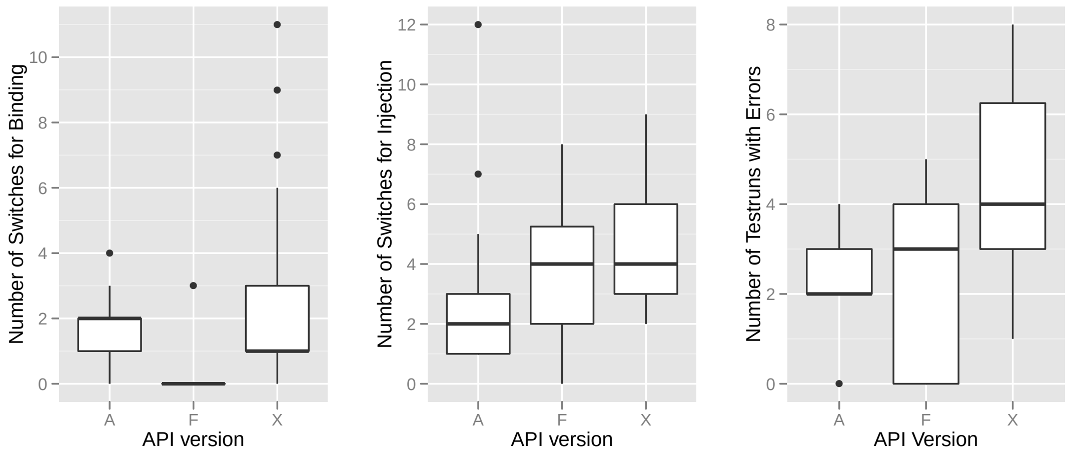


Figure 5.9: Boxplots: (a) Number of switches between classes in IDE for creating a binding, (b) Number of switches for constructor injection, (c) Number of test runs with errors

available constructors and their parameters, but hardly ever happened for creating a binding. When looking at the large time differences between simple and complex tasks for the fluent interface (see figure 5.4), it is highly probable that this is also due to an impact from the number of class switches. While the switching itself doesn't take much time, it is the act of looking up information in the configured class (e.g. parameter types and names of the constructor) that is time-consuming.

A check for statistical significance shows that for creating a binding (figure 5.9(a)), the number of switches with F is significantly lower than for A and X (both p<0.001). In the other hand, for constructor injection (Figure 5.9(b)) the number of switches for A is the lowest (p=0.062 for A<F and 0.002 for A <X). The result that X has equal or more switches than A was unexpected, since with the annotations API users needed to switch a lot because annotations need to be applied directly to the targeted classes. The reason for the XML API needing so many switches was that most users opened every class to copy-paste the class name and often a second time to copy-paste the package/namespace name, because there is no auto completion of class names in the XML. Especially when doing constructor injection, users of F and X needed to switch back and forth multiple times, while with A switching to the class once was often enough.

5.2.7 Error Rate

To check if more errors were made with one API than with another, we compared the number of test runs that were needed to finish all tasks, as shown in figure 5.9(c). All tasks were designed as unit tests, and to show that a task was finished correctly (or to check for runtime errors), users had to run the tests. The median number of test runs with errors was 2 for A, 3 for F and 4 for X. So with the XML API an error happened twice as often as with the annotations API. The number of errors with XML is significantly higher than with the other two (p=0.019 for X>A, p=0.043 for X >F). There is no significant difference between A and F (p=0.485 for A<F).

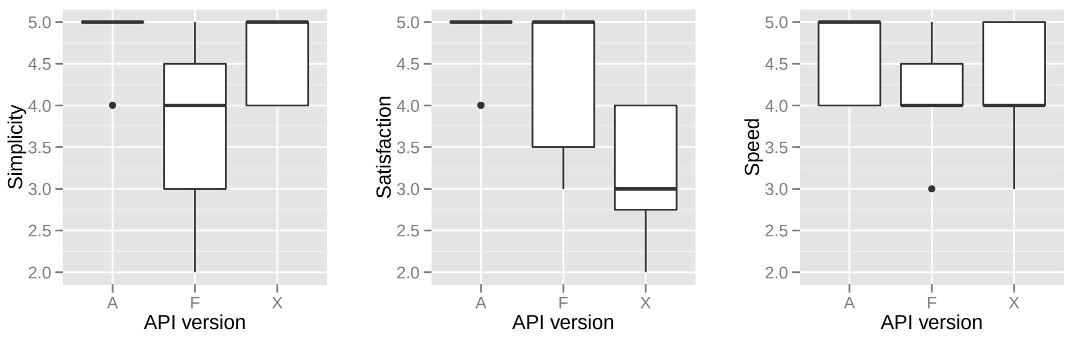


Figure 5.10: Boxplots displaying the results of the post-test questionnaire (1-5, 5=highest): (a) Simplicity, (b) Satisfaction, (c) Speed

The main reason for this is of course the missing auto completion in the XML configuration, and that errors are therefore also not discovered at compile time. Errors happened especially often for longer and more complex words like "Instantiation". Another typical error was that the package/namespace of a class was incorrect, because the classes were sometimes placed in a different namespace/package than the one that was normally used (which was actually prepared exactly to show that this can lead to errors in the XML configuration, while not affecting the other two API variants).

5.2.8 Programmer Satisfaction

After the test we asked each programmer to rate 3 aspects of the used API on a 5 point differential scale as suggested in [142]. The measured aspects were simplicity (Simple ... Complicated), satisfaction (Pleasing ... Irritating/Frustrating) and speed (Fast to use ... Slow to use).

The results are shown in figure 5.10. For *Simplicity*, both A and X were rated with a median value of 5 (the highest rating). F achieved a median value of 4, which is significantly lower than the other two (p=0.016 for F<A and 0.041 for F<X). On the first look it may be unexpected that the XML API was rated significantly higher in simplicity than the fluent interface. But this result also supports the assumptions made in section 5.2.5, that XML has an especially high self-explainability and can be easily understood. With the fluent interface users had difficulties in this area – often users stated that they were unsure which methods can be chained or which must be used consecutively, especially when using more complex functions.

For *Satisfaction*, A and F were rated with a median value of 5, while X was rated significantly lower with a median of 3 (p<0.001 for X<A and 0.014 for X<F). The reason for XML being rated worst is clearly the missing auto completion, both for typing XML element and attribute names, and for typing class names in the XML. Although working with the fluent interface was less simple, it was still more satisfying because nothing needed to be typed in manually.

The smallest differences are shown for *Speed* with a median value of 5 for A and 4 for F and X. There are no significant differences in the speed values, with F<A being the one closest to significance (p=0.060). Especially the fluent interface and XML APIs were rated very equally,

although when looking at the performance the fluent interface actually was almost always significantly faster.

When combining the results for all three rated aspects, the annotations API was rated significantly higher than the fluent interface and XML APIs, which were both rated equally.

5.2.9 Programmer Preferences

In addition to the questionnaire, we asked users about their general opinions on the used API and, in case they had used the other two design patterns, which one they preferred and why. From the 27 test users, 10 users had never used a DI API before, 5 had worked with an annotations-based API, 9 had used an API with a combination of XML and annotations, and 3 a fluent interface. When asked which kind of API they preferred, 9 users had no preferences because of insufficient experience, 12 preferred an annotations-based API, and 6 and API with a fluent interface. None of the users gave a preference for XML.

We asked users for each of the three API variants (also ones they did not use in the Study) what they like and dislike about them. The comments that were given for annotations are shown in table 5.3 (some original comments were kept in German to best retain their meaning). The by far most often mentioned advantage was that annotations can be seen directly in the code that is configured, which makes it easy to understand how the code behaves because you don't need to look somewhere else for the configuration. Users were very excited about this, as can be seen by the statements shown in table 5.3. Other interesting comments on this were: *"Wenn ich jetzt vorhandenen Code bekomme, dann seh ich sofort aha der ist annotiert, und den kann ich soundso verwenden - sonst muss man erst schauen ob es irgendwo eine Konfiguration gibt..."*; *"Wenn alles in einem File gesammelt ist [z.B. bei XML], dann finde ich die Information die ich grade brauche auch schwerer, aber so hab ich genau das was ich gerade wissen will"*; *"Ich hab das Gefühl, dass die Sachen dann einfach "näher am Code" sind. [...] XML ist für mich weit weg vom Code"*. Only few users mentioned other advantages, e.g. that annotations are safer when refactoring code (like renaming parameters) than the other variants, and that no additional file needs to be managed that holds a configuration. The most often mentioned disadvantage was that annotations are very static and don't allow different configurations for the same class, like having one configuration for unit-testing and another one for the production environment.

The comments that were given for the fluent interface are shown in table 5.4. Some advantages were mentioned for both the fluent interface and XML, which is not surprising because they are similar in certain points. These advantages were that the whole configuration is in one place and is therefore easier to overview (which is interestingly the opposite of the most popular advantage of annotations), and that the configuration is independent from the configured code, also allowing different configurations if needed.

For the fluent interface, other mentioned advantages were that the chaining of methods is very comfortable and easy to use once you are familiar with it, and that you can do almost everything with auto completion (unlike with XML). Especially for users that mentioned the former it could often be observed that they were mumbling to themselves in sentences similar to the method chains. For example, when users needed to write the code for constructor injection, which starts like `container.WhenInstantiating<Service>().UseConstructor WithTypes<...`, they mumbled something like "when I am instantiating the class Service,

Table 5.3: Programmer comments on the annotations API

	Comment	Times
+	You can see directly in the class what gets injected (*“ich habs gern [direkt] im Code”*, *“ich habs lieber dort wo der Rest auch ist”*, *“ich seh direkt was injected wird, der Code ist damit viel leichter lesbar”*); this is also an advantage when reading code that was written by someone else; you don’t have to open a different file first	9
+	More convenient than other solutions when you need to annotate a code element that is not directly accessible (e.g. method/constructor parameters)	1
+	Very safe against refactoring	1
+	No seperate configuration file, that additionally needs to be managed (compared to XML)	1
-	Annotations are very static, it is not possible to define multiple configurations (like a separate configuration for testing)	4
-	It can be hard to gain an overview because you have to look into many seperate classes to see the configuration (not everything is in one place)	2
-	You bind yourself strongly to the used framework because you have to write the annotations directly into the classes (invasive)	1
-	You have to deal with the question, in which assemblies/jar files the annotated types can be found and should be loaded from (which is not necessary with the other API variants)	1

then I want to use the constructor with the types ...”. We also see this as a confirmation that once the structure of a fluent interface is understood, it allows coding by building such sentences very intuitively. The main mentioned disadvantage of the fluent interface was that as soon as you need to configure something that cannot be accessed with methods and generics, like constructor parameters, you need to use strings which feels very unpleasant and is not safe to refactoring (unlike annotations, which don’t have this problem). All other disadvantages were centered around the concept of chaining methods, and displayed well the problems we observed during the tests. Users said that it was unclear in the beginning which methods could be chained with which others, that they were unsure if a method chain was complete or if they had forgotten anything, and that longer method chains (4+ methods) were increasingly complex.

For XML (see table 5.5), a mentioned advantage was that the configuration can be changed without recompiling, though on the other hand favourers of the fluent interface mentioned that this was completely unnecessary in most cases. Further, the actual API that is needed in the code is very small and simple: For the DI API it only needed one function, which was creating instances. The main disadvantage was unsurprisingly the missing auto completion, and that due to this there is also no compile-time error checking. A few users noted that this would not have been as much a problem if a schema file had been provided. Some additionally mentioned the problem of refactoring, that most of the time the configuration file is not included in the

Table 5.4: Programmer comments on the fluent interface API

	Comment	**Times**
+	Convenient, easy to read, chaining of methos is simple	3
+	Easy to understand because the whole configuration is kept in one place (compared to annotations)	3
+	It is possible to create different configurations for test and production environment	2
+	Not invasive – no code is added to the classes that are configured, the configuration stays seperate – so the code stays independent of the used DI framework	2
+	Auto completion (code completion, IntelliSense) is available for the API itself as well as the classes that are configured (compared to XML)	1
+	You can get very far with auto completion, whithout having to resort to the documentation – explorative learning is possible because you can go step-by-step from one method to the next (*“Das API ist gut geeignet wenn man wie ich gern einfach drauf los programmiert und schaut was passiert”*).	1
-	Inconvenient when you need to configure something that is not accessible by methods/generics, like method/constructor parameters, because in such a case you need to resort to using strings, which are not safe against refactoring	5
-	It is not exactly clear where the end of a method chain is, i.e. where you should / are allowed to stop, when a chain is complete, or if you have forgotten something	3
-	Longer chains (like with the injection tasks) are complex and hard to read (annotations are much more convenient in this case)	3
-	It is not immediately clear which methods can be chained in which way	2
-	Sometimes you could chain methods in a way that makes no sense	1
-	The chaining of methods needs some getting used to (*“normalerweise verwendet man Methoden nicht so”*)	1
-	Having the configuration in the code makes it impossible to change it whithout recompilation (compared to XML)	1
-	You don’t see directly in a class what gets injected	1
-	May become increasingly complex for larger projects	1

Table 5.5: Programmer comments on the XML API

	Comment	Times
+	The whole configuration in one place (which is more practical especially for large projects, because the configuration is not scattered among a large number of classes, as it would be with annotations)	3
+	The XML structure is easy to understand and read (self-explaining)	2
+	It is possible to change the configuration without recompiling – this allows the user to change the configuration as well	1
+	Not invasive – no code is added to the classes that are configured, the configuration stays separate – so the code stays independent of the used DI framework	1
+	Directly in the code only an API with minimal functionality is required	1
+	It is possible to create different configurations for test and production environment	1
-	No auto completion because of missing schema, which makes creating the configuration cumbersome, and typing errors are not recognized (you have to concentrate a lot to prevent typing errors yourself)	7
-	Also no auto completion of class and package/namespace names, and thus no compile time error checking	3
-	Not safe against refactoring, because everything is defined as strings (*"... und wenn man dann einmal refactored ist alles im Arsch"*)	3
-	Gets complex for large projects, and poses increased maintenance effort	1
-	It is not necessary to configure everything without recompiling, therefore a fluent interface is generally the better choice (*"Freiheit alles zur Laufzeit konfigurieren zu können braucht man eh nie"*)	1
-	You don't see directly in a class what gets injected	1
-	You need to constantly switch between XML and code	1
-	Because of the required configuration file you always have an additional file that you need to deploy together wit your application – if you forget it, the application will not work	1

refactoring process, so refactorings can easily destroy the configuration without directly being noticed.

5.2.10 Impact of Programmer Experience

Like in the first study, we analyzed whether the user's experience had any impact on performance, by comparing the task times with the years of programming experience and the level of domain knowledge. The results are shown in figures 5.11 and 5.12, and are the same as in the first study: there is no significant correlation between any of these values (p>0.1 for all 3 APIs).

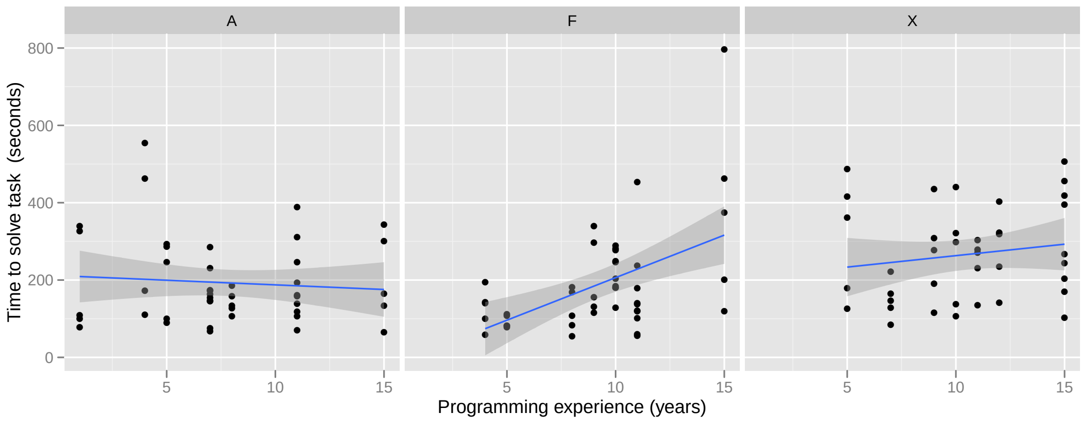


Figure 5.11: Correlation between task times and years of programming experience

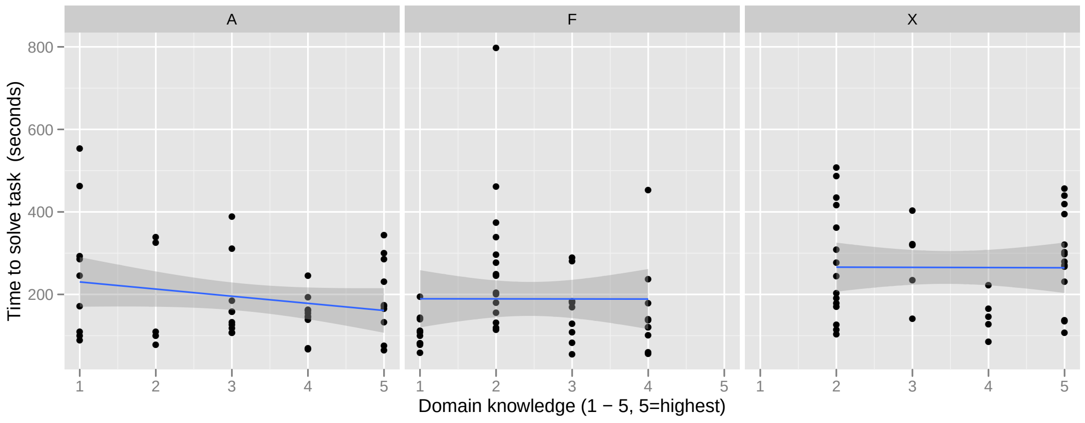


Figure 5.12: Correlation between task times and domain knowledge

This time, we additionally assessed the user's experience with DI frameworks, by presenting the user a questionnaire with a five-point scale for each specific DI framework (see appendix A). We then gave points from 0 to 4 for the ratings and summed them up into a single DI framework experience value. A comparison of this value to the task times is shown in figure 5.13. In this case there is a visible correlation for all three APIs. When evaluated statistically, the results are correlations of -0.23 for A (p=0.06), -0.32 for F (p=0.02) and -0.3 for X (p=0.03). So it is significant for two of the three APIs. This is interesting, as it clearly shows that this "measured" level of domain knowledge is a more reliable factor than the self-assessed one. This may be a typical example of where asking a number of concrete questions that are easier to answer for the user ("how experienced are you with API X?") provides better results than asking only a single question that may be difficult to grasp ("how experienced are you with DI frameworks?") [142].

The DI framework experience seems to mostly affect the times spent in the tutorial. When checking the usage times only (without tutorial), no significant correlation could be found (all

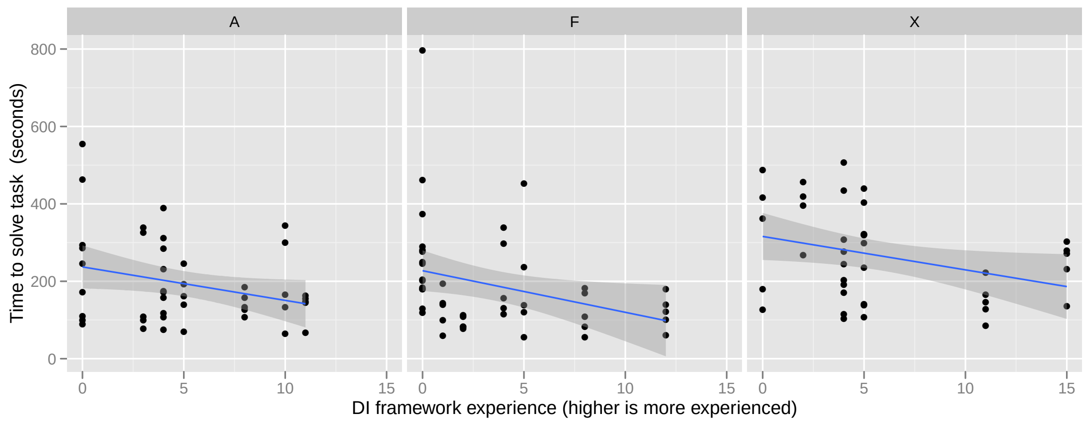


Figure 5.13: Correlation between task times and DI framework experience

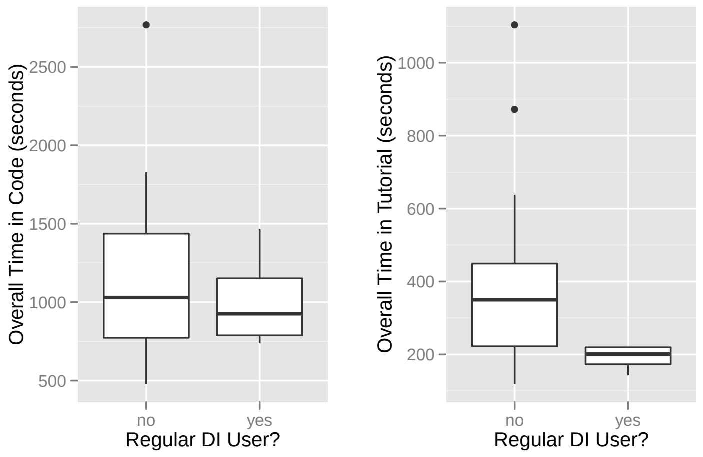


Figure 5.14: Boxplots comparing regular and non-regular DI framework users: (a) overall time spent in code, (b) overall time spent in the tutorial

p>0.05). From this we can conclude that users with more experience in the problem domain tend to spend less time in the tutorial.

We additionally asked users if they were using any DI framework frequently, and compared the overall times spent in code/tutorial as shown in figure 5.14. It shows that users which were using a DI framework regularly were not significantly faster while writing code (all needed about 16 minutes overall). But they spent significantly less time in the tutorial (p<0.001): The median times are 338s for non-regular DI users and 201s for regular ones. This supports our above conclusion. It should be noted though that the significance of these results is limited since only 4 of the 27 users were regular DI framework users.

5.3 Interpretation of Results

Based on the study results we can now present an overview of advantages and disadvantages of the three evaluated design concepts, as well as an analysis which one should be used in which cases. Further we analyze the possibilities for automatically measuring usability and/or making suggestions on the usage of these concepts, which would be necessary for integration into an automated measurement method.

5.3.1 Advantages and Disadvantages

Annotations

The annotations API performed best for almost all measured results. Users were fastest especially when performing more complex tasks, the API was well understood from the tutorial, the learning curve was high, the error rate low, and it was also rated overall best for simplicity, satisfaction and speed. As the main reason for this we see the fact that a considerable amount of information is given simply by the fact where the annotation is placed. So, while with the other APIs the users needed to explicitly say "I want to configure the class Service, and use the constructor with the following parameters...", this was naturally defined with annotations just by the fact that the annotation was placed at this constructor. Therefore also much fewer code elements were needed, e.g. only a single parameterless annotation for defining the injection constructor, compared to two different methods with multiple parameters with the fluent interface. This advantage was stronger, the deeper in a hierarchy the annotation was placed. It was strongest when it needed to be placed at a constructor parameter, which is on the third hierarchical level (class>constructor>parameter). An additional reason for the performance advantage was found in the number of class switches. For complex tasks users needed to look into the classes to create the configuration, which made it necessary for users of the fluent interface and XML to switch back and forth between class and configuration. For annotations this was not an issue, since the class and configuration were at the same place. Only for very simple tasks (operating on the first hierarchical level) the fluent interface had a small performance advantage against annotations (as shown in section 5.2.1), in which case the users could solve the task without ever switching to the configured class.

The simpleness of annotations was also well expressed by the opinions of the users, since a majority of them gave a preference to annotations. What users liked most about annotations, and what was also the most often made statement concerning preferences, is that they are applied directly to the code that is configured. By having the configuration and the element that is being configured at the same place, the code is especially easy to understand, and there is no need to look somewhere else.

There are only few mentioned disadvantages, one being that it gets complicated to get an overview in larger projects because the configuration is spread out across a large number of classes. This was out of the scope of this study, but would be interesting to evaluate in future studies. In general it can be said that from a usability viewpoint, if the use case allows it, an annotations based API should be preferred to fluent interfaces and XML.

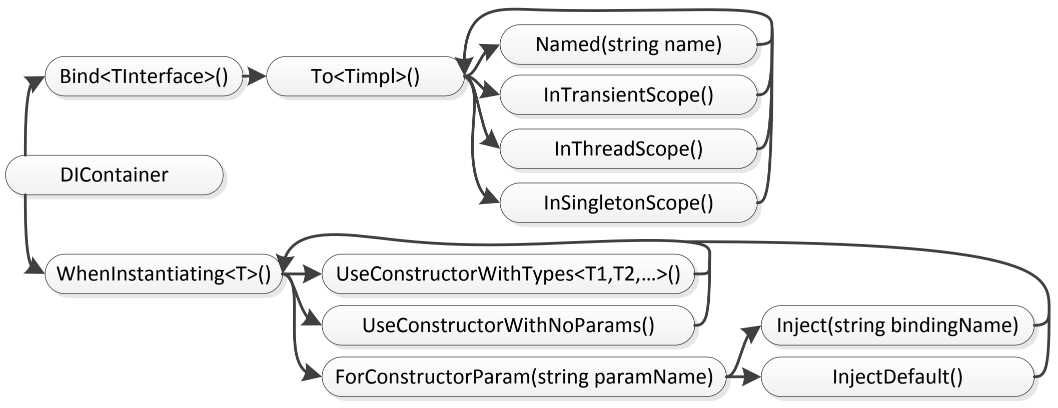


Figure 5.15: Method chaining tree of the fluent interface API

Fluent Interface

The fluent interface showed its main advantages in type safety (especially when compared to XML), as well as in the intuitive way that a method chain can be formed like a sentence in natural language. This enabled users to program in the same way they were thinking, e.g. when thinking "I want to bind ILogger to ConsoleLogger", the code they needed to write was `container.Bind<ILogger>().To<ConsoleLogger>()`.

On the other hand, the chaining of methods was also the main source for complications. While there was no problem chaining only two or three methods, users got confused when more methods were needed, and were wondering whether there was a required order or if a certain method needed to be used or not. Most users seemingly thought that it was not ensured by the API itself that methods can only be called in a valid order (which the API of course did), and that they needed to do that themselves. To illustrate that, figure 5.15 shows the chaining tree for the methods that were available in the study. The arrows show which methods can be called in succession, e.g. to define a named binding with singleton scope, the methods `Bind`>`To`>`Named`>`InSingletonScope` can be chained. In this case, either of the methods `Named` and `InSingletonScope` could be called first, or only one of them could be used alone if the other was not needed. Most often this problem occurred with the more complex chaining tree starting with `WhenInstantiating`. Users were often unsure if `UseConstructorWithTypes` was needed before defining injections for the parameters with `ForConstructorParam`, although the tutorial even contained examples where one was used without the other.

In addition to not knowing if the order is correct or a certain method was needed, users had problems because they were missing some kind of visible ending to a method chain. This is certainly a problem with fluent interfaces, because you can of course just stop writing after any method without the compiler marking any errors. So, an error because of an incomplete method chain would only be found at runtime. In some fluent interfaces we evaluated, an ending method was used to mark the ending of a configuration (like "Start" or "Do"), but with such an ending the chain unfortunately looses readability, since a natural sentence would never end with such a word.

It is to be expected that many of these problems occurred because most of the users had never used a fluent interface before. But rather than that being a threat to the validity of the study, we think that this just shows that fluent interfaces are not widely known, which makes them harder to learn and use.

When looking at these results and the statements of the users, there are some things that can be done to maximize the usability of fluent interfaces: First, users performed much better with short chains (up to 3-4 methods), so if some use cases would require the users to build especially long chains, this could be improved by changing the methods so that the chain is shortened. Second, to minimize the insecurity of users concerning the order and selection of chained methods, the API should whenever possible prevent the user to chain methods that would build an invalid chain. This can be a very difficult task, especially when the API has a large number of chainable methods, but can prevent many potential usability problems.

XML

As a main advantage of XML we identified its understandability and self-explainability. This can be seen when looking at the times spent in the tutorial, where most of the time the code examples alone were sufficient for understanding how to create the XML configuration. We see the reasons for that in the simplicity of the XML language itself (there are just two components that need to be understood, namely elements and attributes, which is far less than in a programming language like Java or C#), as well as the fact that nearly every programmer has already used XML in some way. This makes the users comfortable with XML from the start, which is also shown in the ratings for simplicity, where XML showed good results.

Unfortunately, this cannot compensate the big drawback of missing auto completion. Again it needs to be said that a result of the evaluation of various XML based APIs was that most of them don't offer an XML schema. Even if they offered one, it is not automatically integrated into the IDE, and manual integration is often not done. Although many users criticised that an XML schema was missing, e.g. for Visual Studio none of them even knew how a schema could be integrated (it is a function hidden deep in the options menu). Since we wanted to show the most common use case, we therefore decided that the study should also not contain an XML schema.

If a schema is present, this would definitely improve the performance of XML since it makes creating the configuration faster and prevents typing errors. But it doesn't remove all disadvantages, because there is still no checking of package, class and parameter names. Especially when creating a larger configuration, users rely strongly on copy-pasting the XML elements (this was also observed in the study), which minimizes the problem of typing errors for XML elements, but doesn't change anything about the problems with package/class/parameter names. To prevent this problem, users in the study often copy-pasted the class and package names, which was a significant slowdown, since they often had to switch between classes. These problems are also displayed in the ratings for satisfaction, where the XML API was rated lowest.

From the performance results it can be said that in general one of the other two design concepts should be preferred to XML, as long as the use case allows it. In all performance ratings except for the tutorial, XML performed worst of the three, sometimes even taking twice as much time, because of the mentioned disadvantages (figure 5.5 makes this especially obvious). If XML needs to be used, the study has shown that especially an intuitive and simple XML structure with short and easy to memorize element and attribute names can improve working with the API and prevent unnecessary errors. Further, an XML schema should always be made available.

5.3.2 Suitability per Use Case

The design concept that has been identified as easiest to use is annotations, so it should be the first choice when deciding for a configuration-based API to use. An annotations API can be used whenever the following two conditions are met: First, that the targets of the configuration are classes (e.g. for most of the areas shown in Table 5.1: dependency injection, object-relational mapping, database access and remote communication). Second, when it is possible to statically apply the configuration directly to the classes (which can only be done when the source code of the classes is accessible and can be changed). A fluent interface can be considered alternatively, if the configuration is not too complex (if the configuration of constructors, methods and parameters is not involved). The fluent interface is not limited to only a single configuration, so for simple configurations it could even be preferred to annotations.

The only reason to use XML would be a case where it is absolutely necessary that the configuration can be changed without recompiling. Such a case would for example be logging. To be able to switch logging on and off (or change the log level), it would make no sense to compile the configuration into the program. In all other cases, where Annotations cannot be used, fluent interfaces should be the first choice, since despite being harder to learn the study showed that they provide significantly better usability than XML. Examples where a fluent interface can be the best choice are APIs for unit testing and mocking.

Many existing APIs show that also a combination of multiple design concepts can make sense, e.g. to have a fluent interface and combine it with annotations for more complex class configurations.

5.3.3 Automated Measurement

As described in chapter 3, an important goal of the studies is to find out how the usability of different aspects of an API can be measured automatically. This includes both recognizing aspects and rating their usability. An automated recognition of the three design concepts can easily be done: Annotations and XML can both be recognized by the presence of the corresponding elements. A fluent interface can be recognized by two or more methods of the API being chained together. In case of the presence of XML, a recommendation could be given to use either annotations or a fluent interface instead. In case of a fluent interface, if longer method chains are present, annotations could be recommended as an alternative, if the configuration involves classes (this could be recognized by checking if one of the fluent methods takes a class as parameter type).

For XML, the understandability of the structure is very important. One measurable parameter can be the number of elements and attributes necessary to solve a certain task. Especially when compared to a fluent interface, elements and attributes are mostly equivalent to methods and parameters, not only in their structure but also in the necessary numbers for solving a task. Overall, the usage the XML took about 50% more time than the fluent interface, and should be rated accordingly more complex. Additionally, the fact that more complicated element names were especially hard to memorize could be considered by checking the length of the names. In the study, names with more than 11 letters were perceived especially difficult.

For the fluent interface, an important idicator is the complexity of the fluent method chains. Unfortunately, there is no clear evidence what exactly makes a method chain difficult. Possible measurable properties could be the length of the chain (higher complexity when there are 4+ methods), or the number of chaining possibilities (size of the chaining tree). A reliable combination can probably only be found by trying different properties and checking the correlation to the study results.

The study shows that while the number of code elements (concepts) increases with more complex tasks when using XML or fluent interfaces, it stays mostly the same for annotations, because they are simply placed deeper in the hierarchy. So, this advantage could be measured by the number of needed concepts. While applying an annotation needs more effort than calling a single method, the advantage is simply achieved by the small number of code elements. According to the study results, using one annotation with a single parameter is about equal to two methods with a single parameter each in the fluent interface, and only slightly more complex than instantiating a class.

A more detailed description of how the study results are used for defining high-level concepts for the API Concepts Framework is given later in sections 6.2 and 6.3.

5.3.4 Validity and Applicability of the Study Results

In a study like this, where APIs are used in a rather specific problem domain, questions may arise concerning the general applicability of the study results. To ensure that our study is not only applicable to APIs for dependency injection, we evaluated a much wider range of APIs, for the areas that were shown in Table 5.1. For most of these areas, there are implementations available for all three presented design concepts. The evaluation showed that the general structure, as well as the usability-specific differences between the different design concepts are basically the same for all of these areas. We therefore conclude that our study results are also applicable to at least these areas.

Of course there are also limits to what can be checked with such a study, like how efficient users can work with an API after having used it for several hours, days or weeks of experience. Especially for the fluent interface it can be expected that users would have been able to improve further (see section 5.2.3), although it is unlikely that it would perform better for complex tasks than annotations, so there would be little change to the overall results.

While the study focused on evaluating how programmers work with an API, an additional interesting fact to research in the future would be reading and understanding existing code. This could further strengthen the study results and/or show additional interesting usability aspects. Also a usability evaluation in the context of large projects could bring additional valuable results. Especially the missing support for refactoring in an XML could become a serious problem in this case, and with Annotations the scattering of the configuration over a large number of classes could make it difficult to maintain.

5.4 Study Conclusions

In the presented study we evaluated three configuration-based API design concepts: annotations, fluent interfaces and XML. Users performed best and were most satisfied with annotations, while they performed worst with XML. Annotations proved especially useful when configuring elements deeper in a hierarchy, but can in general only be used when the targets of a configuration are classes. For most other cases, the best choice is a fluent interface, where users performed very well with small method chains, but had often problems when four or more methods needed to be chained. Also, the fluent interface paradigm proved hardest to understand, partly because of the fact that it is the least common one. XML showed its strengths by being very easy to understand and self-explainable, but proved the most cumbersome to use because of its missing auto completion, which was an issue both for XML elements and attributes, as well as for class and package names.

We extracted several properties that could be measured automatically, and could therefore be used for integration into the API Concepts Framework. These results will be utilized in chapter 6 for deriving ways to measure the complexity of HCs.

CHAPTER 6

Elements of the API Concepts Framework

This chapter presents details on measurable properties, low-level concepts, high-level concepts and learning effects, which have been evaluated in the usability studies presented in chapters 4 and 5. In other words, it maps the study findings to concrete elements of the API Concepts Framework. The ZIP usability study will further in short be referred to as [ZIP], and the DI study as [DI].

In our studies the time needed to solve a certain task was used as a main indicator for usability. Consequently we align complexity values to these times in seconds. This means that a higher value always means higher complexity, and therefore lower usability. Further, in accordance to other complexity metrics (e.g. [32, 132]), we don't define a specific unit of measurement for our complexity values. Using any known unit could lead to false assumptions about the results, and creating a new unit would not add any specific value. The question may arise whether using only the task time is enough – [162] shows that the typical usability metrics (task time, task completion, error rate, task satisfaction) all correlate, which strengthens the idea that using only the task time is sufficient.

The API Concepts Framework was published in [171].

6.1 Measurable Properties

This section presents details on measurable properties. For each measurable property, a function c is given to calculate the complexity, as well as an explanation how the function has been created based on the results of the usability studies. Further, a link to the originally identified potential property from section 3.3 is given (if there is one), as well as a reference to the study where the property was identified/evaluated. The mapping of the measurable properties to LCs is defined later in section 6.2.

Some of the measurable properties are based on the number of visible class members and/or overloads. This depends on what is shown in the code completion window [ZIP]. To calculate the number of members for Java IDEs (e.g. Eclipse), all class members including inherited members and overloads need to be counted. In .Net IDEs (e.g. Visual Studio), the code completion window only shows method headers but no overload information, so the overloads must not be counted. Additionally, classes in .Net can have so called extension methods which are not defined directly in the class, but are nevertheless displayed as methods of the class in the code completion window (under the condition that the according namespace is used), so these methods need to be added to the count.

For finding out the "number of members/overloads placed above", the members need to be ordered alphabetically by member name, and overloads need to be ordered first by number of parameters, and then alphabetically, starting with the type name of the first parameter. This ordering was present in all IDEs evaluated in [ZIP].

6.1.1 Number of Members $\boxed{\text{N03}}$ [ZIP]

$$c = n * \frac{1}{3} * \frac{1}{6}$$

n number of visible class members in the corresponding IDE

When searching for a member, the number of other members in the class does not have a large impact overall, but becomes important when the user cannot guess the method name correctly and needs to scroll through the list manually to search for the needed member. Users needed to do that in about one third of all cases when searching for a member. With API V2 they needed 19s for finding a member in such a case, compared to 34s with API V1, which had about 90 more members. This makes about 1/6 second more time per additional visible class member.

6.1.2 Number of Parameters $\boxed{\text{N05}}$ [ZIP]

$$c = \sum_{i=1}^{n} 2.5 * i$$

n number of parameters

We found that there is an exponential increase in the complexity of a method with an increasing number of parameters. We assume the reason is that for each additional parameter, the user must understand its meaning for the method as well as its relation to all other parameters. Users needed 7s, 12s and 28s for using methods with 1, 2 and 4 parameters. The above defined function was built to closely resemble these values. Including the base complexity of 5 for a method call (see section 6.2.3), the function results in values of 7.5, 12.5 and 30 for the same parameter numbers.

The following suggestion is defined: When a method has more than 4 parameters, it should be considered to reduce the number of parameters and/or to provide additional overloads with fewer parameters.

6.1.3 Number of Members with the Same Prefix N06 [ZIP]

$c = n/2$

n	number of visible members with the same prefix (first 3 letters) placed above in the corresponding IDE

The number of members with the same prefix has a much larger influence than the overall number of members. Users needed to find the method `addFile`, but there were many methods with the prefix `add`. In API V1 the method was found 15 positions further down in the code completion window than in API V2, and users needed about 7 seconds more to find it, which is 1/2 second per member placed above in the list. We use the number of members placed above, because users tended to go top-down through the list, and it was especially problematic if the member was placed far down. As prefix we propose to use the first 3 letters of the method, which worked well in our evaluations. Determining the prefix by detecting the first upper case letter would also be possible, but is not reliable if no upper case notation is used.

The measurable property defines the following suggestion: If more than 10 members with the same prefix are placed above the member in question (which means the member would not be visible in a default-sized code completion window without scrolling down), it should be considered to rename or remove some of them.

6.1.4 Number of Overloads N12 [ZIP]

$c = n$

n	number of overloads placed above in the corresponding IDE

A large number of overloads can make finding a specific one difficult, especially because their ordering is often unintuitive. Users took about 1 second longer for each overload placed above in the list (again the overloads placed far down were especially problematic).

Since we found that having a larger number of overloads is better for usability than having only few overloads but with many parameters, we define no suggestion concerning the number of overloads.

6.1.5 Usage of Return Value [ZIP] [DI]

$$c = \begin{cases} 8 & \text{if method has a return value} \\ & \text{and it is used} \\ 0 & \text{otherwise} \end{cases}$$

In [ZIP] several methods were used that had return values which were never needed by the user. For example, the user had to call a method named `addFile` which returned the number of files that were really added. Since not needed, this return value was completely ignored by all users, and seemed to have no measurable impact on usability. On the other hand, [DI] contained method calls where the users needed to use the returned values, which made using the methods significantly slower. In these cases the users were about 8 seconds slower than without using the return value.

This is a property that was newly found in the studies, for which we assign the label R14.

6.2 Low-level Concepts

This section presents a list of identified low-level concepts. We evaluated most of the LCs in [ZIP]. For these the base search and usage complexity values are given, as well as an explanation how these values were assigned based on the study results. Further, the related measurable properties are listed, split into ones that have been evaluated in the [ZIP] and [DI] usability studies, and ones that have not been evaluated in any study yet. An overview of already evaluated LCs, as well as a mapping of LCs to the measurable properties from section 6.1, is shown in table 6.1.

6.2.1 **Class Usage** [ZIP]

This concept represents the action of searching and understanding a class that needs to be used. For several reasons it is the most complex of all concepts: When searching for a class the user has no definite starting point, and most of the time needs to consult a tutorial or documentation, which takes additional time. Further, the user needs to get a basic understanding of how the class is structured to use it efficiently. Class usage always appears in combination with other concepts, where the class name is directly visible in the code (e.g. instantiation, static method calls).

Search complexity = 30
Users took a median time of 30s to find a class.

Usage complexity = 21
Users required a median time of 25s for instantiating a class with a constructor. We found that a majority of this time was spent getting an understanding of the used class (about 21s), and only about 4s for the actual instantiation (which is the complexity value for the concept instantiation, see below).

Measurable Properties
Evaluated in studies: N01 N02 R01 R02
Not yet evaluated: -

6.2.2 **Instantiation** [ZIP]

When using an API, often the first step is that a certain class must be instantiated. So, especially for the classes used in common scenarios, it is very important that they are easy to find and instantiate. Instantiation always appears together with the concept class usage.

Table 6.1: Overview of evaluated Low-level Concepts and Measurable Properties (SC = search complexity, UC = usage complexity)

Concept	SC	UC	Search Properties	Usage Properties
Class Usage	30	21	-	-
Instantiation	0	4	number of overloads	number of constructor parameters
Method Call	10	5	number of members in the class, number of members with the same prefix, number of overloads	number of method parameters, usage of return value
Field Access	10	7	number of members in the class, number of members with the same prefix	none
Annotation	0	12	same as instantiation	same as instantiation

Search complexity = 0
The search complexity is covered by class usage.

Usage complexity = 4
See explanation of class usage (section 6.2.1).

Measurable Properties
Evaluated in studies: N05 N12
Not yet evaluated: R13

6.2.3 Method Call [ZIP]

One of the most typical actions when using an API is calling a method. Often this is done right after instantiating a class. This means that when the developer needs to call a method, he/she has already found the correct class, and is now exploring a very small area of the API. The only exception is when the method is static, in which case the class still needs to be identified, so additionally the concept class usage needs to be applied.

Search complexity = 10
The median search times were 14.5s for API 1 and 12s for API 2 (which had less methods in the class). From these values the complexity resulting from the number of members and overloads according to the identified measurable properties (see section 6.1) needs to be subtracted, resulting in a base value of 10.

Usage complexity = 5
For a method call with 1 parameter, the median usage time was 7s. Since the number of method parameters is a measurable property, the complexity for 1 parameter needs to be subtracted, resulting in a slightly lower value of 5.

Measurable Properties
Evaluated in studies: N03 N05 N06 N12 R11 R14
Not yet evaluated: N09 N10 N11 R03 R04 R06

6.2.4 Field Access [ZIP]

Another typical action when using an API is accessing a field/property. In Java this would mean either getting/setting a public field directly, or using a getter/setter method. In .Net the concept of *properties* is used instead of getters and setters, leading to a somewhat easier to read code, but with equal outcome concerning usability, since Java programmers are as used to using getters and setters, as .Net programmers are used to properties.

Search complexity = 10
Since fields are searched in exactly the same context as methods, the complexity is also the same.

Usage complexity = 7
We found that accessing a field/property is equivalent to calling a method with a single parameter. We therefore align the complexity value accordingly (the rounded complexity for calling a method with a single parameter is 7).

Measurable Properties
Evaluated in studies: N03 N06
Not yet evaluated: R10

6.2.5 Annotation [DI]

The use of annotations is often found in libraries that deal with the serialization of data, like messaging middleware (e.g. WCF) or persistence frameworks (e.g. Hibernate). In both Java and .Net, annotations (or *attributes*) are applied to code elements like classes, interfaces, fields and methods, and can have fields/properties that are assigned in the form of "key=value". As an example for annotating a class, listing 6.1 shows the definition of a database entity class using the Hibernate framework. There are several different ways how an annotation can be used (which is similar in both .Net and Java): Without any values (`@Id`), with values assigned by name (`@Column(updatable=false)`) or with implicitly assigned values (`@OrderBy("litterId")`).

When analyzing annotations from the viewpoint of previous concepts, there are some correlations: Very similar to when instantiating a class, a developer must find and choose an appropriate class for annotation. Also the annotation itself is similar to a constructor call. Assigning a value to the field of an annotation works the same way as accessing the field of a normal class, so this can be handled with the concept field access.

Search complexity = 0
The search complexity is the same as for instantiation.

```
@Entity @Table(name="cats")
public class Cat
{
   @Id @GeneratedValue
   private Integer id;

   @NotNull @Column(updatable=false)
   private Date birthdate;

   @OneToMany(mappedBy="mother")
   @OrderBy("litterId")
   private Set<Cat> kittens;
   ...
}
```

Listing 6.1: Use of annotations by the example of Hibernate in Java

Usage complexity = 12
Users needed more time placing an annotation than normally instantiating a class, because they needed to switch to the class where the annotation was to be added, find the correct place for adding it, and often needed to add package imports to the target class. The median required time was 8s higher than the time for instantiating a normal class, leading to a complexity value of 12.

Measurable Properties
Evaluated in studies: same as instantiation / field access
Not yet evaluated: N13 R12

6.2.6 Interface Implementation

Another concept sometimes found when using APIs is that an interface needs to be implemented. In Java, interfaces can contain methods and constants (though the latter is not important when implementing an interface), in .Net they can contain methods, properties and events.

An interesting example of an API that makes excessive use of interfaces is the open source messaging middleware NServiceBus for .Net. Listing 6.2 shows how endpoints in NServiceBus are configured as clients or servers by using empty interfaces (`IConfigureThisEndpoint`, `AsA_Server`), and how handlers for processing incoming messages are defined by using the `IHandleMessages` interface, which defines the method `Handle`.

There are no studies or guidelines dealing with the usability of interfaces. Again, the process of having to find and choose an appropriate interface is similar to the concept of *instantiating a class*, where an appropriate class for instantiation needs to be chosen. After finding the interface, most programming environments support the automatic creation of stub implementations of all interface members, relieving the developer of the cumbersome work of writing the method headers him/herself. But still, for every interface member, the developer must understand what the meaning of this member is, to be able to implement it correctly.

The question remains how the complexity of each interface member can be rated. Although there are similarities to the concepts *method call* and *field access*, we do not think that these

```
class EndpointConfig :
    IConfigureThisEndpoint, AsA_Server {}

public class EventMessageHandler :
    IHandleMessages<EventMessage>
{
    public void Handle(EventMessage message)
    {
        Console.WriteLine(message.Msg);
    }
}
```

Listing 6.2: Use of interfaces by the example of NServiceBus in .Net

concepts are applicable here, because the process of exploring and finding which member to call is irrelevant in this case. Further research will need to be done to see how interface members can be rated concerning usability, e.g. by carrying out usability studies.

No potential measurable properties were identified.

6.2.7 Inheritance

The concept of inheriting a class is very similar to that of implementing an interface. In both cases, members need to be implemented by the inheriting/implementing class. In the case of inheritance, relevant members are all that are marked as *abstract* in the base class. Additional complexity when inheriting a class comes from the available constructors. If no parameterless constructor is available, the developer needs to explicitly call a constructor of the base class within the constructor of the inheriting class. The combination of choosing an appropriate class for inheritance and calling its constructor can be seen as very similar to the concept *instantiation*, where also an appropriate class needs to be found and then a constructor needs to be called. So, potential measurable properties are all that also apply to the concept *instantiation*.

6.2.8 Exception Handling

Exceptions are a mechanism for handling errors in modern programming languages. When an API raises an exception, the developer needs to catch it to handle the exceptional situation. Little is known about the usability of exceptions, except that they should only be used when exceptional conditions occur (e.g. errors), and not for normal control flow. Whether the exception throwing behaviour of an API is appropriate, will unfortunately be hard to find out in an automated way.

Potential measurable properties are: R07 R08

6.2.9 Package Import

Many of the above concepts involve the usage of a class, and therefore potentially require that a package needs to be imported (`import` statement in Java) or a namespace needs to be used (`using` statement in .Net). In .Net, even a method can require adding a namespace, when it is an extension method that is placed in a separate namespace.

Potential measurable properties are: N08

6.3 High-Level Concepts

This section presents a list of identified high-level concepts. Except for the factory pattern, little is known about the usability of HCs. In [DI] we evaluated some of those yet unresearched concepts, and also identified new ones. An overview of HCs is shown in table 6.2.

6.3.1 Factory [ZIP]

The factory pattern [73] is a widely used object-orient design pattern for the creation of objects. It is quite common in both .Net and Java, as [58] reports. There are actually two related patterns: The "abstract factory" pattern provides a class with which a client can obtain instances of classes conforming to a particular interface or protocol without having to know precisely what class they are obtaining. The "factory method" pattern is a simpler variant, where not a separate factory class is used, but the class that should be instantiated defines the factory method itself. An example for the factory pattern is shown in listing 6.3. It shows how an SSL socket in is created in Java by first acquiring an instance of the `SSLSocketFactory` class and then using the factory's `createSocket` method to create the socket.

The advantage of using factories compared to constructors is that they provide better encapsulation and code reuse, since implementations can be modified without requiring any changes to client code (e.g. when a different concrete class could be returned, which may be necessary for evolving an API [188], but is not possible with a constructor). Factories can also be used to manage the allocation and initialization process, since a factory need not necessarily create a new object each time it is asked for one. However, these advantages mostly concern the developers of an API, but not its users, for which a constructor may be much easier to understand, as [58] shows.

We also found that the factory pattern is harder to understand and use than normal instantiation via constructor. Users took a median time of 44s to instantiate a class using a factory method, but only 25s with a normal constructor. The LCs alone already show this complexity difference (instantiation is less complex than a static method call with a return value), which is a nice proof of their integrity: In addition to the complexity of class usage (21), the complexity for the static method needs to be added (10 search, 5 usage), as well as the additional complexity for the usage of a method return value (8), which sums up to a complexity of 44, corresponding perfectly to the study time. This means that the HC does not need to define any complexity values itself, but only defines the suggestion that normal instantiation should be preferred to a factory method.

6.3.2 Fluent Interface [DI]

Fluent interfaces [71] have only recently become popular, and are inspired by domain specific languages. They try to make use of the natural language by defining methods that are concate-

Table 6.2: Overview of High-level Concepts

Factory	
Recognition	static method calls returning an instance of an API class
Complexity	none – sufficiently rated by the used LCs
Suggestions	use normal instantiation instead
Fluent Interface	
Recognition	existence of fluent method chains
Complexity	for method chains with more than two base class changes +10 for each additional change, +5 when a fluent method is first used, -2 for each method usage
Suggestions	to shorten complex method chains
XML	
Recognition	presence of an XML file
Complexity	15 base complexity for using an XML, 12/20 search/usage complexity for XML elements, 9/9 for attributes, complexity decreased by 33% if XML schema is available, strings additionally evaluated with HC information lookup
Suggestions	if change of configuration without recompiling is not needed, consider using annotations or a fluent interface instead
Config String	
Recognition	recognized by tokens like ";" and ","
Complexity	comparable to XML attributes (not yet evaluated)
Suggestions	none
Class Switches	
Recognition	number of classes that contain implementation code
Complexity	2 switches per additional class, or 5 switches per class when more than 20% of the LCs are used for the first time, 2.5 complexity per switch
Suggestions	none
Information Lookup	
Recognition	strings that are paths or urls, method names containing specific terms suggesting that parameters need lookup
Complexity	8 complexity per string that requires information lookup
Suggestions	prevent the necessity of information lookup where possible

nated to form a readable sentence.

One of the most widely used fluent interfaces is probably LINQ (language integrated query) in .Net. It works with any .Net API that provides an enumerable collection, as well as for querying databases and XML files. In the example shown in listing 6.4, LINQ is used to calculate the average file size of all files in the program files folder. A similar API in Java is jOOQ[1], which defines a fluent language for database access. Other APIs that use fluent interfaces are for

[1]http://www.jooq.org/

```
URL url = new URL(urlString);
SSLSocketFactory factory =
    (SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket)factory.createSocket(url.getHost(),
    443);
```

Listing 6.3: The factory pattern by the example of creating an SSL socket in Java

```
var averageFileSize =
   Directory.GetFiles("c:\\program files\\")
   .Select(file => new FileInfo(file).Length)
   .Average();
```

Listing 6.4: A fluent interface by the example of LINQ in .Net

example mocking frameworks like MOQ[2], and dependency injection frameworks like Ninject.

We found that the main advantage of a fluent interface is that methods can be chained intuitively like natural sentences, which makes it easy for users to work with the API as soon as they have enough experience with it. Disadvantages are that the fluent interface is in the beginning harder to understand because of its dissimilarity with other APIs, and that complex method chains are especially difficult. In [DI] we suggested to measure the complexity of a method chain by its length, but an integration of this idea into the framework did not provide a good correlation to the study results. Instead, a more suitable way of measuring the complexity of a method chain seems to be by the times how often the base class is changed, or in other words how often the selection of methods changes that can possibly be called next in the chain. A simple method chain would be one that always returns the same class, so users can just call multiple methods of this class in a row. A complex method chain would change the class (and thereby the available methods) after every call, or even jump back to the class of a previous chaining step, as it was the case in the complex chaining tree used in [DI]. When the class was changed more than 2 times within a single method chain, users took about 10s longer for each additional class change. Therefore, with n being the number of class changes, we define the complexity as:

$$c = max((n-2)*10, 0)$$

In addition, due to the unfamiliarity of most users with fluent interfaces they needed about 5 seconds more time when they used a fluent method for the first time. Therefore a usage complexity of 5 is added to a fluent method that is used for the first time.

When a method is used in a chain, its return value is always used, which results in additional costs as defined by the measurable property *usage of return value* (see section 6.1.5). In case of a fluent method this is too high since the return value does not need to be stored in a variable. Removing a usage complexity of 2 (an thereby lessening the complexity) for each method call provides the best correlation with the results from [DI].

[2]http://code.google.com/p/moq/

6.3.3 XML Configuration [DI]

The possibility of configuring software via an XML configuration is very common, and offered by a large number of available software libraries (e.g. Spring[3], log4j[4], Hibernate and NService-Bus, to name only a few examples). Despite the popular use of XML configuration, there are no guidelines on how to provide such configuration possibilities.

As an example, an XML configuration for log4j is shown in listing 6.5. In this example, an appender is defined that allows the output of log messages to the console with a special message layout (e.g. to print the logging priority and the current date in addition to the logged message). The appender is then added to the root logger, and the level of output is set to "error".

Compared with other concepts, XML is very simple. It can in principle be understood by knowing just two different concepts: elements (e.g. `<Root>`) and attributes (e.g. `value="error"`) – much less than a programming language. XML is also very easy to read and highly self-explaining, because all elements and attributes are explicitly named. On the other hand, XML has some clear drawbacks when compared to source code: Element and attribute values are just plain text, and except of simple types like integer or enums, their content cannot be evaluated by an editor (e.g. when an attribute contains a class name, an XML editor will not be able to check if this class really exists). Also, code completion features and checking for correct element/attribute names are only available for an XML as long as the corresponding schema file (XSD) is available and known to the XML editor. But we found that many APIs that use XML do not even provide a schema.

Other than that, XML has some structural similarities to code: XML elements can be seen as classes or methods, and attributes as fields or method parameters. Analogously to LCs, we suggest to distinguish search and usage complexity. We found that the best correlation with the study results is reached with the following search/usage complexity values: 12/20 for elements and 9/9 for attributes. If an XML schema is available we propose that all complexity values are reduced by 33%, making the complexity similar to a fluent interface.

The usage of XML also influences users when they are writing code. They were about 15s slower when writing code than the users of other APIs (while using API elements with comparable complexity). We suppose that this comes from the additional effort needed when maintaining both code and XML files. Therefore, for the maintenance of an XML file alone a usage complexity of 15 is added.

Additional complexity with XML comes from the fact that much information needs to be looked up e.g. in another class, like class or method names, since this information cannot be acquired over code completion features. For this complexity, the HC information lookup (see section 6.3.6) is used.

6.3.4 Configuration String

In addition to XML, another concept that is sometimes used for configuration is a *configuration string*. A typical usage scenario is database access, where a connection string is used to connect to a database server. As an example, listing 6.6 shows the creation of a database connection in

[3]http://www.springsource.org/

[4]http://logging.apache.org/log4j/

```
<Configuration status="WARN">
 <Appenders>
  <Console name="Console" target="SYSTEM_OUT">
   <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level
      %logger{36} - %msg%n"/>
  </Console>
  </Appenders>
  <Loggers>
   <Root level="error">
    <AppenderRef ref="Console"/>
   </Root>
  </Loggers>
</Configuration>
```

Listing 6.5: XML configuration by the example of log4j in Java

```
var connection = new
    SqlConnection("DataSource=myServerAddress;Initial
    Catalog=myDataBase;User Id=myUsername;Password=myPassword");
```

Listing 6.6: A configuration string by the example of creating a DB connection in .Net

.Net. Another typical usage of a configuration string can be found in so called property files, which are common particularly in Java (e.g. log4j). In this case, the elements of the configuration string are not separated by semicolons, but by line breaks. We were not able to find any studies or guidelines concerning the usage of configuration strings.

Even more than XML configuration, a config string is very easy to read, because no additional knowledge is needed to understand it. Unfortunately, it is on the other hand very hard to write, since there is no code completion, syntax checking or type checking at all, so the developer is in no way guided when writing it.

Config strings are generally structured like “key1=value1;key2=value2;...”, where a single configuration element would be “key1=value1”, so it should be possible to recognize them by checking for the existence of tokens like “;” or “,”. Concerning complexity, a single element is very similar to an XML attribute, and could therefore be rated with similar complexity.

6.3.5 Class Switches [DI]

We found that switching between classes costs a significant amount of time. One reason for switching is when the implementation code is placed in multiple classes. A typical example is the use of annotations, which must be placed directly in the class that should be configured by the annotation. For every additional class at least two switches are made: one switch to the class, and another one back. Additionally, when the user needs to use a lot of API elements for the first time, the number of switches drastically increases to about 5 switches per class. A reason for this increase is that the user needs to check multiple times if the combination of used concepts over multiple classes is correct.

To detect the presence of new concepts we calculate the percentage of the LCs that are used for the first time, weighed by their complexity values. An increase of switches applies if more than 20% new concepts are used.

One class switch takes about 2.5 seconds. With n being the number of classes and s being the number of switches, the complexity is calculated as:

$$c = (n-1) * s * 2.5$$
$$s = \begin{cases} 5 & \text{if more than 20\% new concepts} \\ 2 & \text{otherwise} \end{cases}$$

6.3.6 Information Lookup [DI]

Another reason why a user needs to switch away from a class, either to look into another class, or even somewhere else (like the documentation), is to look up information that is needed for the API. Most often this concerns the contents of a string parameter which is needed e.g. for a method call or an instantiation. We identified two kinds of information that requires external lookup:

- The first is paths and urls, like the path of a configuration file. In [DI] users of the XML API needed significantly more time to instantiate the DI Container class, because they needed to provide the path of the XML configuration file as a constructor parameter.
- The second is class, method and parameter names, which are especially often needed in configuration-based APIs. In [DI] users were much faster with the annotations API, which was the only one where no lookup of such information was necessary.

An information lookup takes about 8 seconds, which is the additional complexity for concepts where such lookup is needed.

The most difficult part about this concept is the recognition. A path or url can be recognized by its structure rather easily, but recognizing a method or parameter name in a string is more difficult. We suggest to check for method or parameter names having "suspicious" terms in them, e.g. "Constructor", "Method", "Parameter" or "Param". In this case Type parameters are likely to need external lookup. For strings it could be checked if the value is a method or parameter name in another class, which would confirm the suspicion that the parameter requires external lookup.

6.4 Learning Effects

This section presents a list of different learning effects. As explained in section 3.2.3, we define a learning effect by three different reduction values: the reduction after one use (r_1), after two or more uses (r_2), and for multiple uses in the same context (r_c). For a complexity value c the reduced complexity c_r is therefore defined as follows:

$$c_r = c * (1 - r)$$
$$r = \begin{cases} r_1 & \text{for the second use} \\ r_2 & \text{for each use after the second} \\ r_c & \text{for each consecutive use in same context} \end{cases}$$

Table 6.3: Reduction Values of Learning Effects

	1 use	**2+ uses**	**same context**
Search	80%	100%	100%
Usage	20%	40%	60%
HC XML	30%	50%	75%
HC Inf. Lookup	25%	50%	50%

The different learning effects are shown in Table 6.3. In addition to the basic learning effects for search and usage, some HCs define additional learning effects because of special circumstances.

6.4.1 Search Learning Effect [ZIP]

Searching takes especially long when a concept is used for the first time, but as soon as the user knows when to use a certain concept and where to find it, he/she does not need to search for it any more, which is at latest after two usages. In [ZIP], many users did not even need to search any more after a single use, but some still did. We therefore define that the search complexity is reduced by 80% after the first use, and by 100% after two uses.

6.4.2 Usage Learning Effect [ZIP] [DI]

The usage time on the other hand, can of course never reach zero because the user always needs some time to write down the code or at least copy/paste it. Users reached about 40% time reduction when using something for the third time: E.g. for instantiating a class with a constructor, users needed 25s for the first time, but only 15s for the third time. For instantiation with a factory they needed 34s for the first and 21s for the third time. When a concept had to be used multiple times in the same context, and therefore copy/paste was possible, an even higher reduction of 60% was reached.

6.4.3 HC XML [DI]

While the search learning effect for XML is the same as defined above, the measured usage learning effect was higher than for code, because missing code completion support made the first usage especially difficult. So, while using XML is much more complex than code when used for the first time, this difference gets smaller for consecutive usages.

6.4.4 HC Information Lookup [DI]

When having used a concept more often the user knows better what information he/she needs to look up and where. In contrast to other learning effects, copy/paste is not possible in this case, so there is no increased reduction in the same context.

6.5 Conclusions

In this chapter we identified and defined concrete complexity values for 5 LCs, 5 measurable properties, 6 HCs and 4 different learning effects, based on the data of the usability studies. The results are summarized in the tables 6.1, 6.2 and 6.3. Although many potential measurable properties still remain to be researched, the presented results give a good basis for evaluating the API Concepts Framework. Such an evaluation will be presented in the next chapter.

CHAPTER 7

Evaluation and Discussion

This chapter presents a usage example of the API Concepts Framework and investigates how well results calculated with the framework correlate with the results of user studies. Further, limitations of the framework are discussed, and an evaluation with Weyuker's properties [198] is done to investigate the framework's integrity.

As a proof of concept we implemented the framework in C#, including all LCs, HCs and learning effects that were introduced in the previous chapter. Following the structure defined in chapter 3, the implementation of the framework's core functions was very straightforward and also easy to test, since all concepts and measurable properties could be implemented as separate, well decoupled classes. All results that are shown in this and later chapters were automatically calculated using this implementation.

7.1 Usage Example: Constructor Injection

As an example we show a step-by-step calculation for a task from [DI], where users had to do constructor injection for a constructor with two parameters, using two named bindings. We chose this example because it contains a lot of different concepts (both LCs and HCs). With the fluent interface API the following code had to be written:

```
var container = new DIContainer();
container.Bind<ILogger>().To<ConsoleLogger>().Named("console");
container.Bind<ILogger>().To<FileLogger>().Named("file");
container.WhenInstantiating<Service>()
   .ForConstructorParam("consoleLogger").Inject("console");
   .ForConstructorParam("fileLogger").Inject("file");
var instance = container.Get<Service>();
```

First an instance of `DIContainer` is created. Then two different bindings for the interface `ILogger` with the names "console" and "file" are created. After that it is defined that when the class `Service` is instantiated, which has a constructor with two parameters of type `ILogger`,

Table 7.1: Evaluation of Constructor Injection Task with Fluent Interface

Search	Usage	Sum	Concept	Usages	Prev Usages	Description
0.0	12.6	12.6	ClassUsage	1	3	DIContainer
0.0	2.4	2.4	Instantiation	1	3	new DIContainer()
0.0	13.5	13.5	MethodCall	2	4	.Bind<T>()
0.0	13.5	13.5	MethodCall	2	4	.To<T>()
0.0	13.5	13.5	MethodCall	2	2	.Named(String name)
2.2	10.8	13.0	MethodCall	1	1	.WhenInstantiating<T>()
10.2	35.9	46.1	MethodCall	2	0	.ForConstructorParam(String paramName) InformationLookup: +12
10.1	23.9	34.0	MethodCall	2	0	.Inject(String bindingName)
0.0	9.3	9.3	MethodCall	1	2	.Get<T>()
0.0	20.0	20.0	FluentInterface	1	0	Complex Method chain: WhenInstantiating.ForConstructorParam. Inject.ForConstructorParam.Inject (4 class changes)
		0.0	Inf.Lookup			Some concepts require lookup of external information - this should be prevented where possible.
		177.9	**Overall**			

the two named bindings should be used. Finally an instance of `Service` is created using the DI container.

The evaluation of this task is shown in Table 7.1. Each line shows the calculated search and usage cost for a single concept, the number of usages within the task as well as the number of previous usages, and a description. We take a look at how the cost of a single concept is calculated by the example of the method call `ForConstructorParam` (shown at line 7 in the table). This concept is used for the first time, so there is no learning effect from previous usages.

First the search and usage complexity is calculated: The base search complexity for method call is 10. The complexity for the measurable property number of members is 0.2 (rounded) since the class has only 3 public members. For the other search properties it is 0 because there are no overloads and no members with the same prefix. So the resulting search complexity is 10.2. The usage complexity is calculated from the base value of 5, the number of members (complexity is 2.5 for one member), and the usage of the return value (complexity 8), which makes a total of 15.5.

After that the high-level concepts are evaluated, and a *fluent interface* is recognized because of the fluent method chains. This HC alters the complexity of the method call: -2 for less impact of the return value, but +5 for the first usage of this fluent method. Further, the HC *information* *lookup* is detected because the method parameter is a parameter name from another class. The cost for this is 8. This results in an overall usage complexity of 26.5 for the method call.

Since the method is used twice in the code, the cost for a second usage in the same context needs to be calculated: In case of search complexity there is no additional cost because of the reduction by 100%. The usage complexity for the second usage is reduced by 60%, the cost for information lookup by 50%, and there is no additional cost for the fluent interface any more, which results in 9.4 for the second usage. With the two usages together, the overall usage cost is 35.9, as is shown in Table 7.1.

This example also shows how a suggestion is displayed in the results as part of the description: For the HC *information lookup* (last line in the table) a suggestion is given that lookup should be prevented when possible.

7.2 Correlation with Study Results

For an evaluation of the validity of the results, we extracted 126 measured median values from [DI]. Since a majority of the of the LCs, measurable properties and learning effects have been derived only from the results of [ZIP], the data from [DI] can serve as a control sample for these parts. The measured values include times for initialization (instantiating the `DIContainer` class), creating bindings, defining injections, and getting instances from the DI container. For each of these categories, times for search and usage were analyzed separately. For each value we then calculated the complexity with our framework. To evaluate the data statistically, we use the *Pearson correlation coefficient*, which is a measure of the linear correlation (dependence) between two variables, where 1 is a total positive correlation, and 0 is no correlation.

There is a high correlation of 0.93 between the measured and calculated values (see figure 7.1). It is even higher when looking at the usage values alone (0.96), which means that our framework provides very exact results for measuring usage complexity. For search complexity, the correlation is 0.87, which is still very good, especially when taking into account that the two studies used different learning approaches (exploration in [ZIP], tutorial in [DI]). All correlation values are statistically significant ($p<0.001$).

There is unfortunately no separate control study available for checking the high-level concepts, but the main goal of this article, which was proving the feasibility of the introduced framework, is nevertheless satisfied.

To strengthen the validity, we additionally compare our framework with results from two other API usability studies, which were carried out by another research group. These are the only external studies where sufficient data (the used API, task code and results) are available for comparison: [58] presents a study with several tasks analyzing the factory pattern. The times for the “Sockets Task” are 7min 41s for using a constructor, and 16min 5s for using a factory. The corresponding results of our framework are 79 and 152, which correlates nicely to the study results (in both cases a factor very close to 1:2). [184] presents a study about the influence of method placement on usability. For the “Email Task” users needed 1 vs 11 minutes with good vs bad method placement, and for the “Web Authentication Task” they needed 2 vs 15 minutes. Our measurement results are 152 vs 218 and 209 vs 256. The gap between the measured times is larger than between the calculated values, but we see the reason for that in details that were

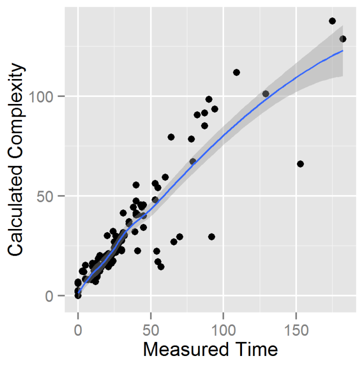


Figure 7.1: Correlation of measured and calculated values for [DI]

specific to the study: The influence of bad method placement was especially high because the users were forced to learn the API purely by exploration (which is not the case in a real world scenario, where users would have consulted a documentation or the web).

7.3 Limitations

There are two main reasons why the correlation of the search values is not as good as the correlation of the usage values, which can also be seen as the two main limitations of the framework with its current list of concepts and measurable properties: The first is the naming and placement of the API elements (e.g. method names). The second is that the framework does not take into account the complexity of the documentation.

7.3.1 Naming and Placement

[126] shows how complex naming in computer programs can be. It says that "Programming is a sophisticated intellectual process that combines aspects of natural language with the regular structure of formal mathematical thought". Natural language is highly diverse and unstructured, and is therefore very hard to analyze in an automated way. It requires a deeper understanding of the cognitive model programmers apply when naming code elements, which is a complex research topic on its own.

When searching for a certain element in the API, much depends on whether the element is named and placed like the user expects it to be. In [ZIP] some cases showed this clearly: To open an existing ZIP file, the API contained a method named `Open`, but some users strongly believed that the method would be named `Read`. In such a case, although there was not a large number of methods, it could sometimes take a long time for the user to find the differently named method, and it was very obvious users were thrown off course by the fact that the method they expected did not exist.

An example for problematic method placement was also found in [ZIP]: For one task users had to update an existing zip file. While it was clear for all users that a method named something like `Update` needed to be found, most users searched in the wrong place at first. While they expected the method to be part of the `ZipEntry` that should be updated, the method was actually placed one layer above, directly in the `ZipFile` class.

To our knowledge no research exists on how to measure naming and placement of API elements for usability purposes, but if measurable properties for this are found in the future, they can be easily integrated into our framework.

7.3.2 Documentation

Most of the time developers use tutorials, code examples and other documentation (also including anything that is found on the web) to learn how to use an API. This of course makes the complexity of the documentation to an important API usability characteristic, as also seen in [13]. The results of [DI] show that the framework is already able to show the complexity to a good degree when a tutorial is used for learning. But especially for more complex tutorial chapters, where users first had to understand how different API elements are connected to each other, there was a large difference between the measured and calculated values, leading to the lower correlation of search complexity values. Therefore, integrating documentation into the measurement approach is a major topic of future research. While some ideas in this direction are presented in [13], to our knowledge there exists no method yet for measuring the usability of documentation.

CHAPTER 8

A Middleware Optimized for Usability: The Application Space

One of the motivations for writing this thesis was the realization that, especially in the area of middleware communication and concurrency, there are actually no APIs available that are really easy to use. Take for example WCF (Windows Communication Foundation), which is .Net's prime communication solution, and started out with the goal to make remote communication easier. It is surely an improvement over low-level TCP socket communication, but as soon as things like publish/subscribe and other forms of asynchronous and parallel communication are required, WCF gets very complex and cumbersome to work with. For a long time, we (the Space Based Computing Group at TU Vienna) believed that space-based middleware [140] could solve such usability problems because of a paradigm shift, which allowed programmers to understand communication in a more intuitive way. Spaces can provide elegant solutions even for complex problems, like load balancing [59] or load clustering [116]. But the solutions still proved too hard to understand for programmers that had never been in touch with space-based computing. We therefore wanted to find out what influences the usability of APIs, and based on the findings create a middleware solution that is flexible and easy to use. The result is a middleware that still takes ideas from space based computing, but is stripped down to the most crucial communication functions. It is called the Application Space.

This chapter describes the Application Space, the motivations behind it, and how its usability has been evaluated. First, section 8.1 presents general use cases to evaluate the usability of middleware. After that, section 8.2 introduces the Application Space, as well as the technologies which it was built upon. In section 8.3 the usability of the Application Space is evaluated and compared to other middlewares, using the API Concepts Framework.

8.1 Use Cases for Evaluating the Usability of Middleware

Since usability can never be evaluated without a specific context of use, we need to define use cases which can then be implemented and analyzed. We have identified different layers of use cases for middleware, by rating the complexity of the use cases – while the use cases on the same layer are similarly complex, there is a significant difference in complexity between each layer:

- Layer 0: installation/setup (preparing the middleware for usage)
- Layer 1: communication, concurrency, asynchrony
- Layer 2: anonymity (i.e. lookup/discovery mechanisms), time-independence (communication partners don't have to be online at the same time)
- Layer 3: replication, fail-over

Layer 0 corresponds to our definition of *setup complexity* (see section 3.1.1), which is not targeted by the API Concepts Framework. We will therefore analyze this aspect only descriptively. An ideally simple setup would e.g. only require to add a single library reference to the code, a complex one would e.g. require the installation of additional software.

Our main goal for the Application Space is to provide easy to use solutions for problems on layer 1. While this may seem simple, we will show that even on this layer solutions can become quite complex with existing middleware, so there is a lot of potential here. And we firmly believe that only a middleware that is easy to use for simple problems, can also be easy to use for complex ones. So the Application Space doesn't aim to solve all complex problems, but rather to build a platform with proven usability for handling simple problems, on which middleware for more complex problems could easily be built upon (like the Peer Model, see section 8.2.3).

In our usability comparison we will therefore focus on layer 1. The following are use cases of this layer. For each use case also a short real-world example is given. The examples are taken from the FFG Bridge project *AgiLog* (Agile Logistics), within which the AppSpace was proven and tested with real-world scenarios.

1. **Simple message transfer:** A simple message (e.g. text) is transferred in one direction between two communication partners. For example: Mobile devices send GPS data to a server, where it gets stored in a database.

2. **Request/Response:** A server that receives a request from a client sends back a response, which is then processed by the client. For example: A client requests GPS data for a certain vehicle and date to display on a map, the response contains this data.

3. **Request with multiple responses:** A server sends back multiple responses to the request of a client (e.g. multiple messages about the progress of a long running operation, before in the end the result is sent to the client). For example: A client requests gps data from

the server, and wants to display a progress bar to give the user visual confirmation on the progress of the operation, before displaying the gps data on a map.

4. **Concurrent communication:** Multiple incoming messages shall be processed concurrently, ideally with a configurable degree of concurrency. For example: The incoming gps data of multiple mobile devices shall be processed concurrently on the server.

5. **Sequential communication:** Multiple incoming messages must be processed sequentially. For example: The incoming data of a single mobile device must be processed in the order of arrival, so that e.g. incremental GPS data (that only contains the difference to the previously sent GPS point) is stored correctly.

6. **Communication with error handling:** A client that sends a message must be able to react to communication errors related to this message, as well as to errors that are thrown by the server. For example: When the server cannot be reached, the client wants to retry sending the message until it is successful. When the server throws an error while processing the message, the client wants to send an email to inform the customer and/or the helpdesk staff about the problem.

7. **Publish/Subscribe:** Multiple clients subscribe to a service, to receive a series of events that are published by this service. They unsubscribe from the service when they don't want to get any more events. For example: To keep the position of vehicles up to date on a map view, clients subscribe to a service that publishes an event whenever the position of a vehicle is updated. To reduce network traffic, a client only keeps the subscription active only as long as the map view is really active (i.e. unsubscribes when the user leaves the map view).

8.2 The Application Space

The Application Space (or short: AppSpace) is a framework for distributed, asynchronous and parallel communication, developed by XCoordination[1]. It has been created and steadily improved within the past 6 years, while working on this thesis. Several articles have already been published about the AppSpace in the German developer magazines dotNetPro [194–197] and Visual Studio One [164–166], and talks about it have been held at German and Austrian developer conferences. The AppSpace is available for free under the GNU General Public License version 2 (GPLv2), and can be downloaded from the open source project hosting site CodePlex[2]. Tutorials and code examples are also available there.

[1]http://www.xcoordination.com

[2]http://xcoappspace.codeplex.com

8.2.1 Base Technologies

The AppSpace is inspired mainly by two technologies/paradigms: space based computing, and Microsoft's Concurrency & Coordination Runtime.

Space Based Computing

The idea of space based computing was first created by David Gelernter in the 1980s [74]. He introduced a coordination language called *Linda* which operates on an abstract computation environment called *tuple space*. Distributed processes coordinate each other by writing and reading tuples to/from the space. So the space acts like a shared blackboard – everyone can put something into the space, or get notified when something has been added by another process. The space is responsible for coordinating access to the data.

The Space Based Computing Group[3] at the Institute of Computer Languages at TU Vienna has been researching space based middleware solutions for many years. A result of this research is XVSM (eXtensible Virtual Shared Memory) [10, 27, 42, 43, 48, 97, 140, 206], which further extends the traditional Linda tuple space model with very useful concepts. In Linda a space is completely unstructured (i.e. there is no order of items), and the only coordination mechanism is template matching, which makes some problems rather difficult to solve, like when there is a required order in which items should be taken from the space. XVSM solves this problem by structuring the space into *containers*. Entries in a container can not only be coordinated with template matching, the coordination mechanism can be freely replaced, so for example the container could instead act like a queue, stack or dictionary. The XVSM space implementation is also called *coordination space*. It provides operations to write, read, take and delete items to/from a container, as well as to get notified when e.g. an entry is added. Further, XVSM supports transactions and is easily extensible with *aspects* (a term borrowed from aspect oriented programming). For many use cases, XVSM has proven to be very flexible and provide elegant solutions, for example for agent coordination [117], load balancing [59] and load clustering [116].

Despite of that, we found that a problem with XVSM is that the paradigm is just too far away from the technologies that most developers are accustomed to. This makes it hard to explain to them, what the major advantages of space based middlewares actually are. The AppSpace wants to eliminate this problem, by providing an API where developers can find similarities to things they already know, making it easier to accept the new paradigm. Of course the major advantages of XVSM should not be lost, but rather just reduced to their most basic form. In our point of view this includes:

- Processes should be decoupled. Further, communication should not happen by calling methods (which enforces things that should be avoided, like synchronism), but by asynchronously transferring data (e.g. messages). In its basic form, this means that all communication should be asynchronous, and processes don't have to wait for each other.

[3]http://www.complang.tuwien.ac.at/eva/

- Many actions can happen concurrently, and these actions must be coordinated, if they conflict with each other. Strongly simplified, this means that the execution and coordination of concurrent and exclusive actions must be supported.
- Notifications must be supported, so that processes can get notified about something. This feature is also known as publish/subscribe.

Because the new solution should be a simplification and generalization to the XVSM coordination space, and because it should generally make the implementation of any kind of distributed application easier, the name *Application Space* was chosen.

Concurrency & Coordination Runtime

At the time the implementation of the AppSpace was started in 2009, features for concurrency and asynchrony in .Net and Java were very rare and not very sophisticated. But to implement the desired functionality such features were absolutely necessary for the AppSpace. Therefore, a rather unknown, but very promising .Net library from Microsoft was found, called the *Concurrency & Coordination Runtime* (CCR) [155]. The CCR was initially built for solutions in the robotics area, and is part of Microsoft's Robotics Developer Studio[4]. But because of its unique concurrency features and good performance, it also gained some recognition outside of robotics, and was released as an independent framework, which can be downloaded and used for free. The following code example shows the basic API elements of the CCR:

```
var dispatcher = new Dispatcher(0,"default");
var queue = new DispatcherQueue(dispatcher);
var port = new Port<string>();
port.Post("Hello, World!");
Arbiter.Activate(queue, port.Receive(msg => Console.WriteLine(msg)));
```

The `Dispatcher` is the CCR's own highly configurable threadpool. Its constructor allows defining the number of threads – if 0 is defined, then the number of threads will equal the number of processor cores. The dispatcher gets the tasks that need to be processed from a `DispatcherQueue`. The central and most important element of the CCR is the `Port`. In its basic form, it is like a typed queue. Items that are posted to it using the `Post` method are stored in FIFO order. What is unique about the port is, that a specific action can be mapped to it, which is executed for each item that is posted to the port (or that is already in the port). This mapping is created using the `Arbiter` class as shown in the example. In this case, every item that is posted to the port will be printed to the console. As soon as an item is posted to the port, a work item consisting of the posted item as well as the action to be executed is queued into the associated dispatcher queue. The dispatcher then takes work items from the queue and processes them whenever one of its threads is free. The important part is that actions are automatically executed asynchronously and concurrently, running on the threadpool of the specified dispatcher. This makes it much easier to write asynchronous and concurrent applications than with the low-level .Net `Thread` and `Threadpool` classes.

What makes the CCR even more valuable are the numerous coordination features around the port class. Given a list of multiple ports, it is possible to asynchronously wait for messages

[4]https://www.microsoft.com/robotics

to arrive on all ports (`Join`), or on only one of them (`Choice`). Further, with a feature called *concurrent/exclusive/teardown groups*, it is possible to define that among a group of ports, items of a specified port should be processed completely exclusive (meaning no other items can be processed at the same time), while items of other ports are allowed to be processed concurrently. What is especially elegant about this feature is, that no explicit locking is required and no threads are blocked at any time – all coordination is handled asynchronously by the CCR in the background.

The AppSpace makes use of the CCR as a library, and integrates it directly into its API.

8.2.2 The Application Space API

This section introduces the API of the AppSpace. The AppSpace was built following the findings and results that have been described in this thesis. The most important of them are:

- Since class usage is the most complex of all concepts, the goal is to have an API that requires as few classes as possible. In detail, this means to minimize instantiations, static method calls, inheritances and interface implementations.
- Default method overloads should be provided for standard usage scenarios.
- In a standard scenario, developers should not be required to do any special configuration. Configuration that is absolutely necessary (like the tcp port on which a space instance is running) should be as simple as possible.
- If special configuration is required, either annotations or a fluent interface should be used, but XML should be avoided.
- To maximize the learning effect, the same concepts should be reusable in different situations.
- Specialized API concepts for beginners and experts should be considered as described in [142], to further optimize the learning curve, and make the first experiences with the AppSpace as simple as possible.

The most important concept of the AppSpace is the *port*, which is taken directly from the CCR, along with its abilities for asynchronous and concurrent message processing. In addition to the functions already provided by the CCR, the AppSpace makes the port remotable, in other words messages can not only be posted to a port locally, but also from remote locations. Ports can even be sent along with transferred messages (comparable to the endpoint of a queue, or the URL of a container in XVSM), and messages posted to them will always be transported back to the original owner of the port instance. This makes the port an extremely flexible concept that can transport any kind of message in any direction.

On top of the CCR ports, the AppSpace defines the concept of *workers*. This concept allows creating messaging contracts that are similar to e.g. service contracts in WCF. When comparing the AppSpace with a service based communication middleware, calling a specific method of a service is similar to posting a message of a specific type to a port. As interfaces are used to bundle multiple service methods into a single service contract, analogously worker contracts are used to bundle multiple message types. And as a service implements the methods defined

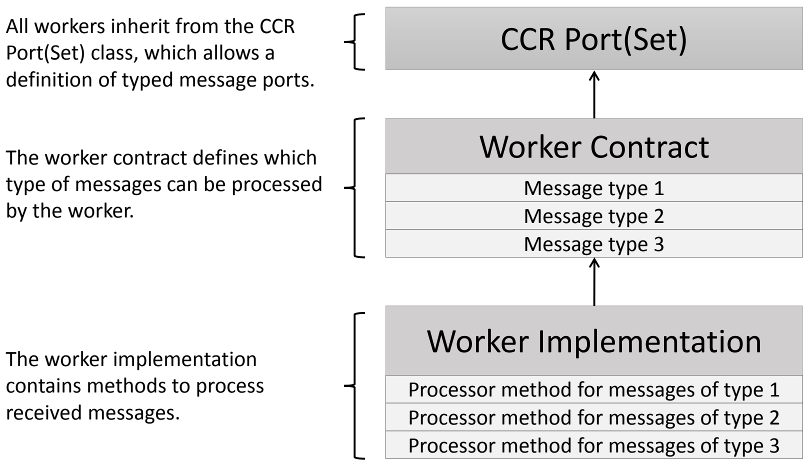

All workers inherit from the CCR Port(Set) class, which allows a definition of typed message ports.
CCR Port(Set)
The worker contract defines which type of messages can be processed by the worker.
Worker Contract
Message type 1
Message type 2
Message type 3
The worker implementation contains methods to process received messages.
Worker Implementation
Processor method for messages of type 1
Processor method for messages of type 2
Processor method for messages of type 3

Figure 8.1: The structure of workers in the Application Space

in the contract interface, a worker implements processing methods for the messages that are posted to the worker's ports. In code, a worker contract is a class that inherits from either `Port` or `PortSet` (the latter is a collection of multiple ports). A worker again inherits from the contract class, and defines a processor method for each message type. Figure 8.1 illustrates this relationship.

Basic Usage Scenarios

The AppSpace was designed so that in simplest usage scenarios only a single class needs to be instantiated, which is called `XcoAppSpace`. This class allows hosting ports and connecting to remote ports. Unlike many other middlewares like WCF, the `XcoAppSpace` class is used independently of whether a communication partner acts as a client or as a server, so there is no requirement of strictly defining such a role, and the developer only needs to deal with a single class on both sides of a communication (i.e. it acts as a "peer").

The following code shows how the AppSpace is instantiated, and how a port is hosted that can be reached by other AppSpace instances:

```
var space = new XcoAppSpace("tcp.port=8000");
space.Run<string>(msg => Console.WriteLine("Msg received: " + msg));
```

A simple configuration string is used to define on which tcp socket port the AppSpace is running (in this case 8000). The `Run` method creates a port instance (not to be confused with the tcp port) and maps the port to the given delegate method, which means that all incoming messages will automatically be processed using this method (which in this case prints the messages to the console). The generic parameter `string` is the type of message that can be handled by the port.

The code for a client that sends a message looks very similar, with the difference that Connect is used instead of Run, to connect to the port of other AppSpace instance:

```
var space = new XcoAppSpace("tcp.port=0");
var port = space.Connect<string>("address:8000");
port.Post("Hello, world!");
```

This time the value 0 is used for the tcp socket port, which means that simply a random port number will be used. When calling Connect, the message type and address of the port that is running on the remote AppSpace need to be specified. The port instance that is returned by this method can be used just as if it were a local port. When calling Post, the message is automatically transported to the remote AppSpace, where it will be processed as specified.

So a very simple message sending example with the AppSpace requires really only 5 lines of code. This basic example doesn't even require the earlier explained concept of workers. To make the first example a developer comes in contact with as easy as possible, we chose to follow the advice in [142] and create a beginner API without workers, which basically consists of the *Run* and *Connect* methods. Not only simple one-way communication can be solved like this, but also sending a response, or even multiple responses. The only thing that needs to be done for that, is to use a custom message type instead of string. For example, if as a response to the string message an integer should be returned, the message type could look like this:

```
[Serializable]
public class Request {
   public string Message { get; set; }
   public Port<int> ResponsePort { get; set; }
}
```

To get a response, the client sends a so called *response port* to the server together with the message. The server can then simply post messages to this response port to send back an answer. The only requirement for custom message types is that they are marked with the Serializable attribute (which is required by the underlying BinaryFormatter that is used for serialization by default).

Workers

While using workers adds a small amount of complexity, it also has several advantages: By defining worker contracts, multiple message types can be grouped like methods in a service. When clients connect to a worker (meaning the connectivity to the worker is checked, and a local proxy object for accessing the worker ports is created), they also get access to all of the worker's ports at once, and don't need to deal with connecting to every single port separately. Further, workers allow defining whether messages are processed concurrently or exclusively, by leveraging the CCR's *concurrent/exclusive/teardown group* feature.

A chat application is often used by our research group as a simple example to illustrate the features of XVSM. It is also a good example for workers: A chat worker could define two different message types, one for receiving a message (string), and another one for getting the current list of messages (GetAllMessages). The worker contract would then look like this:

```
public class ChatWorkerContract : PortSet<string, GetAllMessages> { }
```

```
[Serializable]
public class GetAllMessages {
   public Port<List<string>> ResponsePort { get; set; }
}
```

The worker implementation inherits from the contract and defines processor methods for the message types:

```
public class ChatWorker : ChatWorkerContract
{
   [XcoExclusive]
   void Process(string msg) { ... }

   [XcoConcurrent]
   void Process(GetAllMessages msg) { ... }
}
```

This example shows how message processors can be either marked as concurrent or exclusive. All incoming messages of processors marked with `XcoConcurrent` will be processed concurrently, using the CCR dispatcher. All that are marked with `XcoExclusive` will be processed completely exclusive, meaning that no other message for any of the worker's ports can be processed at the same time. This is very convenient here, because adding a chat message is a write operation, so it needs exclusive access to underlying resources (e.g. a list of strings that is hosted by the worker). On the other hand, getting the list of messages is a read operation, so concurrent access of multiple clients at the same time is no problem.

This example also shows the parallels to XVSM, and how the principles of space based computing were generalized and simplified for the AppSpace. A worker can be seen as a resource that optionally can hold a state, and therefore is able to implement the behavior of space container.

Publish/Subscribe

Another feature that is required in many remote communication scenarios is publish/subscribe. The chat application again is a good example, where clients should be notified when new messages are posted. For that purpose, the AppSpace provides a class called `XcoPublisher`. This class handles the two predefined message types `Subscribe` and `Unsubscribe`, and can easily be added to any worker. To do this, the two message types need to be added to the worker contract, and an `XcoPublisher` instance needs to be added to the worker implementation:

```
public class ChatWorkerContract : PortSet<string, GetAllMessages,
    Subscribe<string>, Unsubscribe<string>> { }

public class ChatWorker : ChatWorkerContract
{
   XcoPublisher<string> publisher = new XcoPublisher<string>();

   [XcoExclusive]
   void Process(string msg) {
      ...
      publisher.Publish(msg);
   }
   ...
}
```

The subscribe message simply contains a port which is then stored in the XcoPublisher, and whenever a chat message is published, which is done using the Publish method, it is automatically posted to all registered ports. This is again a similar solution to how notifications work in XVSM.

Error Handling

Error handling is very important, but complicated for asynchronous operations. In this case the AppSpace borrows again from the CCR, which defines a very elegant feature called *causalities*. With this, a special port is defined to which exceptions are posted in case of errors. The AppSpace makes this feature remotable, so that it also works when an error is thrown while processing a message in a remote AppSpace instance. The exception port can simply be provided by using the method PostWithCausality instead of Post:

```
var exceptionPort = space.Receive<Exception>(
   ex => { ... dosomething ... });
port.PostWithCausality(mymessage, exceptionPort);
```

Not only exceptions thrown during processing can be captured this way, but also communication exceptions, e.g. when the space where the remote port is hosted is not reachable.

The examples until now have shown that the port as a central concept is extremely flexible and can be used in all kinds of situations. This is the beauty of the AppSpace: Though there is only a small set of API concepts and features in comparison to other middlewares like WCF, their flexibility allows handling different communication scenarios easily and with a high learning effect, precisely because everything is done with the same concepts.

Advanced Configuration

In addition to a simple configuration string, the AppSpace provides a fluent interface with advanced configuration options. Annotations were not an option in this case, because the configuration needs to be flexible, and involves mostly classes where the user has no direct access. The detailed configuration possibilities can be found on the AppSpace homepage, and will not be handled in detail here. The following example shows some of the features:

```
XcoAppSpace space = XcoAppSpace.Configure
   .UsingService<XcoBinarySerializer>().AsDefault()
   .UsingService<XcoJsonSerializer>().WithName("json")
   .UsingService<XcoTCPTransportService>()
      .OnPort(8000).AsDefault()
   .UsingService<XcoTCPTransportService>()
      .WithName("mobile").WithSerializer("json")
      .OnPort(8001).WithAuthenticationKey("securekey");
```

This is again a real world example from the AgiLog project. In this configuration, two TCP transport services are used on two different ports, one of them using a binary serializer for message serialization, and the other a JSON serializer. While the former is responsible for communication between server applications, the latter handles incoming connections from mobile devices, which only support the JSON format. Since the mobile device connection needs to be globally accessible over the internet, it is secured by an authentication key.

Additional features include for example different transport services like WCF, MSMQ, Jabber and Azure, as well as security features for authenticating worker access.

8.2.3 Facilitating the AppSpace for Complex Coordination Middleware

One of the ideas behind the Application Space is that it shall serve as a basis for future space implementations. Such implementations can then focus on their core tasks like coordination and transactions, and leave things like concurrency and asynchronous communication for the AppSpace to deal with. Recently a new programming model was invented by the Institute of Computer Languages, called the *Peer Model* [115], which is inspired by the principles of both AppSpace and XVSM. A first implementation for the Peer Model called *PeerSpace.Net* was created based on the AppSpace [154], and the experiences for facilitating the AppSpace for this task were reportedly very positive.

8.3 Usability of the AppSpace Compared to Other Middlewares

It is finally time to make use of the API Concepts Framework, to compare the usability of the AppSpace with other middleware solutions. For comparison we chose two representative and well known middleware solutions that follow different paradigms: One is the service-call based middleware WCF, the other one is the enterprise service bus NSericeBus.

For each middleware we evaluated the scenarios described in section 8.1, by first implementing a working solution and then measuring the usability with the API Concepts Framework. Here we describe three of the scenarios, which give a good idea of the middleware's overall usability:

- simple message transfer
- request with multiple responses
- publish/subscribe

Based on the implementation code for each scenario, we evaluate *interface complexity* with our framework. Further, we evaluate *implementation complexity* with the following two metrics:

- **Cyclomatic complexity (CC)** [132] gives an estimation of the overall complexity and understandability (a low value means low complexity and good understandability). It measures the complexity of control flow, and is calculated by the number of decisions that can be taken in a method. We use the same definition of CC as it is used in the tool NDepend [179], which is as follows [5]: The CC of a method is 1 + {the number of the following expressions found in the body of the method}: if, while, for, foreach, case, default, continue, goto, &&, ||, catch, ternary operator ?:, ??. Additionally we count inline anonymous methods (lambda expressions), recognized by =>. We sum up the calculated values of all methods to get a single cyclomatic complexity value.

[5]http://www.ndepend.com/Metrics.aspx#CC

- The **logical lines of code (LLOC)**, in other words the number of logical statements, are used to estimate the size of the solution. This is not only everything that ends with ";" – combined statements must also be considered, like `list.add(new Message(...));` which consists of two logical statements (an instantiation and a method call). Additionally we count class and interface definitions, method headers, annotations and field/property definitions as independent statements. While structural code like that is normally not taken into account by LLOC, we think that a fair comparison is only possible when it is also considered.

In section 3.1.1 we proposed to use some additional metrics for measuring implementation complexity, like the maintainability index [146, 193], but these metrics only get relevant for larger solutions. Since the implementations presented here are comparatively small, the metrics would have no significance, so we don't use them here.

The following sub sections present the concrete evaluation results. Since the listings and usability evaluation tables take up a lot of space, they have been separated from the description and can be found in appendix D.

8.3.1 WCF

The Windows Communication Foundation, in short WCF, is .Net's primary solution for any kind of remote communication. It follows the service call paradigm, meaning communication is done by calling methods of a service, which is represented by a class/interface. WCF can be configured either directly in code, or with an XML configuration file. In the following examples we will use a code only configuration, since it is easier to compare directly to the AppSpace (the used classes are very similar to the XML configuration elements, so we don't expect WCF to have any special advantage or disadvantage from this decision).

Setting up WCF is very easy, since there is actually nothing to do, despite of adding a reference to the assembly `System.ServiceModel` which is contained directly in the .Net Framework. The reference is even added automatically if the Visual Studio IDE is used to create the WCF service class.

All listings and evaluation tables can be found in appendix D.1.

Simple Message Transfer

To implement this scenario, first a service contract needs to be defined, which is annotated with `ServiceContract` and `OperationContract`. On the server side, this contract needs to be implemented, and the implementation needs to be annotated with the `ServiceBehavior` attribute. For publishing the service, a `ServiceHost` needs to be instantiated and configured. On the client side, a `ChannelFactory` needs to be created to get access to the service. On both sides, the binding that is used for communicating (in this case a `NetTcpBinding`) must be specified.

The result oft the calculation of interface complexity with our framework is a value of 503.2, which is quite large for a simple task. The major factor that causes this high complexity value is the large number of classes that are involved: 6 different classes need to be either instantiated or annotated. One way to reduce the number of classes would be to provide better defaults, so that e.g. the `NetTcpBinding` would be the default if no binding is specified explicitly.

The implementation complexity values are CC=3, because there are only three methods and no control flow structures, and LLOC=23.

Request with Multiple Responses

Although the scenario may seem simple, it is quite complicated to solve with WCF. We see the main reason for that in WCF's service call paradigm – a method simply is not meant to return multiple results asynchronously. Because of that, the solution feels a bit like a "workaround". To support returning multiple result messages, a *callback channel* is required. For this, first a `CallbackContract` needs to be defined, which is a separate interface in the contract. This interface needs to be implemented on the client side. To provide it to the server, an `InstanceContext` needs to be created, and a `DuplexChannelFactory` (which is a different class than the one used in the first scenario) needs to be used to open the connection. On the server side, the callback channel needs to be acquired using the `OperationContext` class. Further, the `ConcurrencyMode` of the service needs to be set to `Reentrant` (otherwise messages cannot be returned to the client while the service method is still running).

The interface complexity value for this scenario is 745.9. Again a lot of different classes (8) are required, which could be improved by providing better defaults. The fact that the concurrency mode needs to be set to reentrant is also very hard to understand, and could for example be simplified by making this the default value in case the service has a callback contract defined.

The implementation complexity values are CC=4, which is 1 higher than for the first scenario, because an additional method needs to be implemented for the callback contract, and LLOC=37.

Publish/Subscribe

Since WCF doesn't provide any publish/subscribe functionality out of the box, this feature needs to be implemented partly by hand. The same concepts as for the second scenario are required for that, which makes interface complexity only slightly higher, with a value of 772.3 (the only difference is that the `OperationContract` attribute is required 2 additional times). The functionality is implemented as follows: The service provides two methods for subscribing and unsubscribing. In the `Subscribe` method the client's callback channel is retrieved and stored in a dictionary under an ID that is specified by the client. To unsubscribe, the client calls the `Unsubscribe` method, which removes the callback channel with the specified ID from the list of channels. Within the service's `Notify` method, the list of callbacks is simply gone through in a loop to publish a message to all subscribed clients. Clients that are not available any more (recognized when the callback throws an exception) are removed from the list.

While the interface complexity is not much higher than for the second scenario, there is a significant different in implementation complexity: CC=9 because multiple additional non-trivial methods need to be implemented for the publish/subscribe functionality. LLOC is also significantly higher with a value of 58. So, to make this scenario any easier, the most important requirement for WCF would be to provide publish/subscribe functions that are integrated directly into the API.

8.3.2 NServiceBus

NServiceBus is the most popular service bus middleware for .Net. It calls itself “the most developer-friendly service platform for .Net”, which, next to having good tooling support, could also be interpreted as a statement for having good usability. NServiceBus uses message queues for communication. By default MSMQ (Microsoft Message Queueing) is used, but other message queue implementations like RabbitMQ and ActiveMQ are also supported. The API of NServiceBus is based strongly on implementing interfaces. For example, there are interfaces to configure whether a node is a client or server endpoint, to specify that something should run when the node starts up, or to handle messages of a specific type. In addition to that, things like the message queue configurations and addresses of publishers are configured in an XML configuration file.

Setting up NServiceBus included both positive and negative experiences. The framework can be automatically installed with the .Net package manager NuGet, which is directly integrated into Visual Studio. NuGet not only downloads the NServiceBus libraries and adds references to the projects, it also automatically adapts the compile configuration, and creates an endpoint configuration class. The custom compile configuration is needed because the code is not running in its own executable, but is hosted by an NServiceBus node. Having this configuration done automatically is very pleasant and removes a significant amount of complexity when setting up a new project. What needs to be done by hand though, is the installation of additional software that is required by NServiceBus. This includes the underlying message queuing middleware (e.g. MSMQ, which is not installed by default in Windows), as well as the database RavenDB[6], which is used for persistence.

The most negative experience when setting up NServiceBus came not until the code was run for the first time, which resulted in an error stating “No endpoint configuration found in scanned assemblies”. According to internet resources this seems to be a very common error, and can have many different reasons, like that the assembly was compiled with the wrong .Net Framework version. In our case, the problem was that the assembly was set to compile for “x86” platforms instead of “Any CPU”, but this was very hard to find and took a significant portion of the overall implementation time.

All listings and evaluation tables can be found in appendix D.2.

Simple Message Transfer

To implement this scenario with NServiceBus, endpoints need to be defined for both client and server. This can be done using the interfaces `IConfigureThisEndpoint` and either `AsA_Client` or `AsA_Server`. Since a template for this is created automatically, this step is quite simple. In the contract a message type needs to be defined. This type must implement one of the interfaces `IMessage`, `ICommand` or `IEvent`. The documentation doesn’t make it

[6]http://ravendb.net

clear what the functional differences between them are, especially because none of them define any members. In the case of our example we use `IMessage`. Transferring simple types like `string` is not supported, so a message contract always needs to be defined.

On the server side, a handler for the message can be defined by creating a class that implements the interface `IHandleMessages<>`. This interface defines the method `Handle`, which is called when a message of the defined type arrives. On the client side, a class needs to be defined that is run at startup, which can be done by implementing the interface `IWantToRun-WhenBusStartsAndStops`. To send a message, access to an `IBus` instance is required, which gets automatically injected when a public property of type `IBus` is defined. This bus instance provides a method called `Send` which can be used to send a message to the server. By default, the server's address is a message queue with the same name as the server assembly (in this case `MsgTransfer.Server`). An interesting detail about the `Send` method is that the message parameter is of type `object`, implying that any kind of object can be sent, which is actually not true, as stated above. Although this fact is currently not evaluated by our usability framework, we already identified this as a potential factor that influences usability (see section 3.3, R04), which should be evaluated in the future.

When referencing NServiceBus, an XML configuration file with some basic configuration elements and attributes is automatically created, but an example as simple as this one can also work completely without any XML configuration. We therefore don't rate any complexity for it.

The interface complexity value for this scenario is 496.7. Like with WCF, the main factor for this high value is the large number of 7 required classes, since nearly everything is specified with interfaces.

The implementation complexity values are CC=2 and LLOC=20. CC is very low because there are only two methods required: the one that sends a message on the client side, and the one that handles it on the server side. There are no `main` methods required because the code is hosted in NServiceBus nodes.

Request with Multiple Responses

Since a service bus is built for asynchronous communication, implementing this scenario with NServiceBus is much easier than with WCF. The required concepts are nearly the same as for the first scenario. In this case three different message types are needed (a `Request` is sent to the server, `Progress` messages are replied while processing, a `Response` is replied in the end), which all implement the `IMessage` interface. On the server, a handler class implementing `IHandleMessages<Request>` is required, and on the client side two handlers for progress and response messages are needed, which are implemented with the same interface. On the server side, when a message has been received, the `Reply` method of the `IBus` instance is used to send messages back. The endpoint configurations on both sides and the startup on the client side are the same as in the first scenario. So, compared to scenario 1, only a single new concept was needed, which is the method `Reply`.

The interface complexity value is 576.3 for this scenario, which is only slightly higher than for the first scenario, since classes like `IMessage` and `IHandleMessages<>` can be reused multiple times.

The implementation complexity values are CC=4 and LLOC=38.

Publish/Subscribe

If service buses have been invented for a certain scenario, then it is publish/subscribe. It is therefore expected that this scenario doesn't require much more effort than the previous ones. This proves to be mostly true for NServiceBus, although some additional concepts are required. This time the roles of client and server are reverted – the client implements a message handler, and the server runs at startup to publish some messages. One difference for the server is that the endpoint now needs to be configured using the `AsA_Publisher` interface (instead of `AsA_Server`). The reason why there are two different interfaces for these two roles, or why the roles have to be explicitly specified at all, is not completely clear. Another difference is that the message class needs to implement `IEvent` instead of `IMessage`, for which the reason is unfortunately also not clear. To publish messages, the method `Publish` is used.

The biggest difference though is that on the client side the source of published messages needs to be specified. This is done via XML configuration, which is read by the node during startup, and lets the node automatically subscribe to the configured events. The XML configuration contains the name of the contract assembly, the message type and the name of the publisher endpoint. This makes three different values which all require information lookup, and need to be configured without auto completion and compile time checking. Although configuration in code is also possible, it seems to be rather complicated, and is apparently not the way it should be done by default. An additional problem with this solution is that the exact point in time where the client subscribes and unsubscribes cannot be controlled, since it is bound to the client's lifetime, so the requirements of the scenario are actually not completely fulfilled. This needs to be taken into account when comparing the usability results with other APIs.

The interface complexity value for this scenario is 685.2. The increase compared to the previous scenarios is mainly because of the required XML configuration, since there seems to be no easy way of adding publisher endpoints in code.

The implementation complexity values are CC=2 and LLOC=28 (including the lines of XML configuration). These values are quite low which shows that NServiceBus supports this scenario very well (though that doesn't mean the API itself is easy to handle).

8.3.3 Application Space

Compared to WCF and NServiceBus, the communication paradigm of the AppSpace is something in between. It has worker contracts and the ability to connect to ports/workers, which is similar to how a WCF service is defined and a connection to a service is established. In the other hand, it supports messaging and asynchronous communication similar to a service bus.

Setting up the AppSpace is very simple, like with NServiceBus it can be done by installing the package with NuGet. After that no additional setup actions are required.

All listings and evaluation tables can be found in appendix D.3.

Simple Message Transfer

Message transfer is very easy with the AppSpace, as already shown in section 8.2.2. On both sides first an instance of the class `XcoAppSpace` is created. On the server side, the `Run` method is called to create a port and map it to a function that is called for incoming messages. On the client side, the `Connect` method is called to connect to the port, and `Post` is used to post a message which is then sent to the server. Since only a message of a simple type (`string`) is transferred, no explicitly defined contract is necessary.

The interface complexity value for this scenario is 169, which is much lower than that of WCF and NServiceBus. The main reason for this is that only a single class (`XcoAppSpace`) needs to be used directly, and can be used on both client and server side. Although the class `Port` is also used, it is never used directly (e.g. for instantiation), and therefore doesn't influence the complexity.

The implementation complexity values, which are CC=3 and LLOC=10, show that not only very few concepts, but also very little code is needed.

Request with Multiple Responses

For this scenario, a contract is required to send response ports to the server together with the message. The message needs to be annotated with the `Serializable` attribute. Although this attribute doesn't directly belong to the AppSpace, we counted it as a concept, because from a developer's point of view it is actually used like it were an AppSpace API element. On the client side, the two ports need to be created and mapped to functions, using the `Receive` method. On the server side the `Post` method is used for replying, in the same way that it is used on the client side for sending.

The interface complexity value for this scenario is 338.1. The increase of complexity compared to the first scenario is mainly because of the message contract, which requires the classes `Port<>` and `Serializable`. But despite of that, the complexity is still significantly lower than for NServiceBus and WCF. Compared to WCF, the AppSpace is just much more suitable for asynchronous messaging. Compared to NServiceBus, there are several reasons why the AppSpace requires fewer concepts and has better reuse: There is no distinction between client and server endpoints, only a single method is used for sending messages (`Post`) instead of two (`Send` and `Reply`), and only a single message type needs to be implemented instead of 3 because the response ports don't need their own message types.

The implementation complexity values are CC=5 (because of the 3 lambda expressions for handling port messages) and LLOC=22.

Publish/Subscribe

This scenario requires the use of workers (see section 8.2.2), since the classes for the publish/subscribe functionality need to be hosted within a worker. Therefore, this time a worker contract is needed, which inherits from `PortSet`, and defines three message types. One is the message to be published (`string`), and the other two are predefined message types for subscribing and unsubscribing to/from events (`Subscribe<>`/`Unsubscribe<>`). The worker implementation on the server side inherits from the contract, and needs to define a field of type `XcoPublisher<>`, which handles the publish/subscribe messages, as well as a processor method for the `string` message, which needs to be annotated with the `XcoConcurrent`

Table 8.1: Usability Evaluation Results Overview

Scenario	**Middleware**	**Interface Complexity**	**Implementation Complexity**	
			CC	**LLOC**
Msg Transfer	WCF	503.2	3	23
	NServiceBus	496.7	2	20
	AppSpace	169.0	3	10
Req / Multi Resp	WCF	745.9	4	37
	NServiceBus	576.3	4	38
	AppSpace	338.1	5	22
Publish/Subscribe	WCF	772.3	9	58
	NServiceBus	685.2	2	28
	AppSpace	565.9	4	25

attribute. Within this processor method, the `Publish` method is called to publish the message to all subscribers.

On the server side the worker is run using `RunWorker`, and on the client side the connection is established using `ConnectWorker`. Further, on the client side a subscription port is created on which published messages are received. This port is sent to the server by posting a `Subscribe<>` message. Unsubscribing works in the same way, by sending an `Unsubscribe<>` message.

The interface complexity value for this scenario is 565.9. The increase in complexity compared to the previous scenarios mainly comes from the three additional classes (`XcoPublisher`, `Subscribe`, `Unsubscribe`) that are required for this scenario. There may still be room for improvement in this case, e.g. by integrating the functionality tighter with the Port/PortSet classes, and offer it as methods instead of completely separate classes. Nevertheless, the result is still the best of all three middlewares.

The implementation complexity values are CC=4 and LLOC=25.

8.3.4 Comparison of Results

A comparison of the results of the three middlewares is shown in table 8.1. It clearly shows that the AppSpace is the winner in interface complexity as well as LLOC in all three scenarios. CC and LLOC are more or less the same for all middlewares and scenarios, with one exception: Since publish/subscribe is not supported by WCF, additional logic had to be implemented in the code which clearly shows in increased CC and LLOC values. With the presented scenarios, it can be said that CC values of 5 or less and LLOC values of 40 or less show that the middleware provides all the functionality required for the scenario, and that the only thing the code needs to do is use this functionality. On the other hand, CC>5 and/or LLOC>40 indicate that additional functionality needed to be implemented.

The AppSpace shows the biggest advantage in interface complexity for the first scenario, where the complexity is only about one third compared to that of the other two middlewares. The reason for this is that the AppSpace provides a very simple beginner API without workers and contracts, which the other two middlewares don't. The results confirm that such a beginner API makes sense to make the first steps as easy as possible. Another very positive result is that the AppSpace beats NServiceBus even in one of its key functionalities, which is publish/subscribe. This is even more impressive when taking into account that the NServiceBus solution for this scenario doesn't meet the requirement of flexible subscribing/unsubscribing.

Up to this point we compared the three middlewares from the viewpoint of developers that don't have experience with any of them. But a very interesting question is of course: If one already knows e.g. the basics of WCF or NServiceBus, does it still pay off to learn and use the AppSpace? This question can be answered with the API Concepts Framework, by changing the number of previous usages for the analyzed concepts. For the message transfer scenario with WCF, the interface complexity value is 257.8 with 1 previous usage, and 158.4 with 2+ previous usages for all concepts. For NServiceBus, it is 257.7 with 1 and 160.6 with 2+ usages. This shows that even if a developer already knows a bit of WCF/NServiceBus (1 previous usage), it still pays off to use AppSpace instead. Even for experienced developers (2+ usages) the effort of using the AppSpace (complexity 169) won't be much higher than for WCF/NServiceBus (158.4/160.6), but will pay off greatly after having learned how to use the AppSpace.

So concerning the AppSpace, the evaluation showed that the we successfully created a middleware solution with good usability, which performs well compared to other middlewares, even for developers that already have some experience with these other middlewares. Concerning the API Concepts Framework we can say that it successfully mastered its first real-world application, and provided very useful results, which were also helpful for reasoning about possible usability improvements. Findings from such evaluations, e.g. that the number of classes for simple examples often seems to be too high, could also flow back into future versions of the API Concepts Framework, and be integrated there e.g. as suggestions or high-level concepts.

8.3.5 Evaluation with the Cognitive Dimensions Framework

A recent student project [159] at the Institute of Computer Languages also investigated the usability of APIs, using the cognitive dimensions framework [37, 39]. The evaluated APIs were WCF, the service bus implementation MassTransit, the AppSpace, as well as a first prototype implementation of the peer model (called peer space, see also section 8.2.3). The results are interesting both because of the evaluated APIs, and because of the used cognitive dimensions framework, which apart from our own framework is currently the only usability measurement method that allows a thorough investigation of API usability. In contrast to our framework, the cognitive dimensions framework is an inspection method where one or more human evaluators rate the usability of an API by analyzing 12 different dimensions as defined in [39]. In the project each API was analyzed using these dimensions, and inspecting similar scenarios as the ones that were defined in this thesis, as well as two more complex scenarios with master/worker and split/join coordination. For each cognitive dimension an "optimum" value was deduced (e.g. the consistency of an API should be high, and the premature commitment required to work with an API should be low), which an API should aim to reach.

The results of the project are shown in figure 8.2. To compare the results, the divergence between the defined optimum value and the API's concrete value was calculated for each cogni-

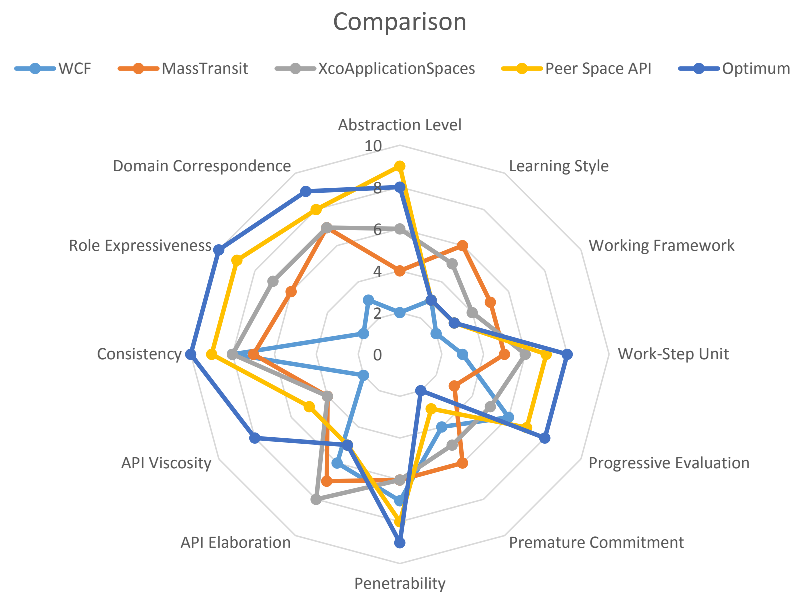


Figure 8.2: Results of the middleware comparison based on the cognitive dimensions framework [159]

tive dimension, and the resulting values were summed up. The worst of all APIs was WCF, with a divergence of 41 points, followed by MassTransit (which is similar to NServiceBus) with 39 points. The AppSpace reached a clearly better rating with 30 points. Problems when using the AppSpace were mainly found with the complex scenarios, where the AppSpace didn't provide sufficient mechanisms for a simple and straightforward implementation. The best result with only 10 points divergence was reached by the prototype peer space API, which builds on the AppSpace and provides better support for complex coordination tasks.

The overall ratings for WCF, AppSpace and the service bus coincide with those of the API Concepts Framework (i.e. AppSpace was clearly the best, followed by the service bus, and WCF on the last place), which is another proof for the integrity of our framework.

CHAPTER 9

Conclusions

Usability is an important quality attribute not only for graphical user interfaces, but also for APIs. But existing methods are difficult to use and require a lot of resources. Further, they don't provide objective and comparable results, and depend strongly the experience of the evaluators. In this thesis we presented an automated framework for the measurement of API usability, called the API Concepts Framework, which aims to provide a solution to these problems.

In section 1.4 we defined 5 goals for the thesis. We will now check whether these goals were fulfilled, and what has been done to fulfill them:

Identification of Factors Influencing API Usability
In chapter 3 we investigated related work and identified 22 different factors that influence API usability. From these factors we derived 26 potential measurable properties, from which we evaluated a selected few with two usability studies. We did a detailed analysis of the study results in chapters 4 and 5, utilizing different statistical data evaluation methods, and successfully identified several measurable properties that have a proven influence on API usability.

Definition of an Automated Framework for Measuring API Usability
In chapter 3 the API Concepts Framework was defined, and in chapter 6 concrete elements of the framework were described. The framework consists of low-level concepts, high-level concepts, measurable properties and learning effects. This modular structure makes it highly extensible, so new concepts, properties and learning effects can easily be added in the future.

Proof of the Correctness and Suitability of the Measurement Method
In chapter 7 we evaluated our framework, by comparing calculated results with existing API usability studies. The results are very promising, with a correlation value as high as 0.93. As a proof of the framework's correctness/integrity, we analyzed it with Weyuker's properties, which is a set of properties that are desirable for software metrics. We showed that our framework satisfies the properties very well, with the only possible improvement that relationships between concepts could be taken into account more strongly.

Identification of the Limitations of the Measurement Method
Through the detailed analysis of the study and correlation data, we identified two main limita-

tions of the framework with its current set of concepts. One is that the understandability of API names, which is an important usability factor, is not yet taken into account. The other is that the usability of the documentation is not explicitly evaluated. Still, even without these two factors the framework already shows very good results. Both factors are areas of future work.

Evaluation of a Middleware API Optimized for Usability
Finally, in chapter 8 we presented the Application Space, a middleware that is optimized for usability. We used our framework to compare the usability of the AppSpace with other prominent .Net middleware solutions for different usage scenarios, and the AppSpace showed clear advantages in all cases. The API Concepts Framework not only helped comparing the middlewares, but also to identify reasons why an API was more difficult to use (e.g. because of a large number of required classes) and to find possible areas of improvement.

We can therefore conclude that all goals of the thesis have been reached, and that a new automated and objective measurement framework for API usability has been developed, which is the first of its kind, and has been proven to successfully measure the usability of APIs.

9.1 Future Work

The present thesis was an important first step into the area of automated API usability measurement. However, a lot of work still remains to be done in the area, towards the goal of being able to measure the usability of any kind of API reliably, taking all kinds of usability factors into account. The following points are part of this future work:

- An important point is of course the further evaluation of the two main limitations of the framework. One is the evaluation of the naming of API elements. Possible ideas how to measure naming need to be analyzed concerning their reliability and integrability into the framework. One such idea is for example: It could be analyzed if words in the textual description of a use case can be mapped to the code, e.g. when looking at the ZIP study: For the description "create a new zip file, add a single file and then save it", the API elements `ZipFile.Create(...)`, `ZipFile.AddFile(...)` and `ZipFile.Save(...)` would map well. This way it could also be possible to find code on the wrong abstraction level, like the use of streams when creating files [37]: e.g. if the description is "create a file containing the text xxx", then the word "stream" in a class would get a negative rating because it appears nowhere in the description. It could even be possible to do such an evaluation based on existing unittests, if *behavior-driven development* [180] is used, which involves textual behavior descriptions for each tested use case.
- The second limitation we identified is the usability of the documentation. It needs to be further evaluated how this aspect can be measured, and how it can best be integrated into our framework. Since documentation is much less structured than source code, and can come in various formats, an important part of this will be to acquire an understanding of semantic methods. Possible ideas towards documentation usability are: The method could search for the API elements in the documentation that are used in the code, and

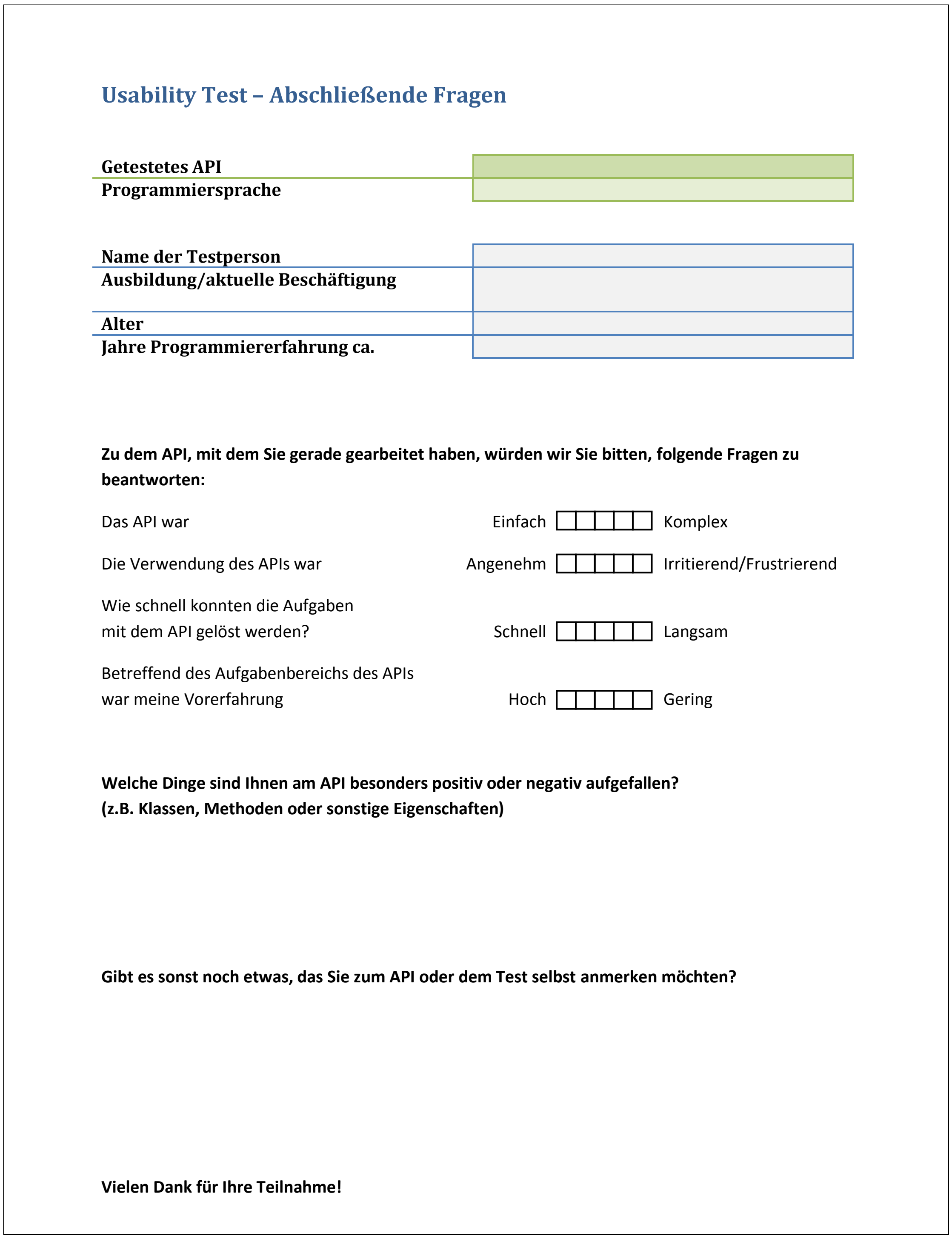
then measure e.g. how hard this section is to find, how big it is (the size of the section may indicate how difficult the concept is to understand), and if there are code examples. Further, the "distance" between required API elements in the documentation may be an indicator for the overall learning effort.

- In this thesis we presented two usability studies where we evaluated numerous potential measurable properties concerning their influence on usability. But, many of the potential properties we identified in section 3.3 still remain unchecked. It is therefore of course an important part of future work to also evaluate these properties, to gain a more complete picture of the factors that influence API usability.
- Further, the APIs of only very few problem domains have been evaluated until now. One that is especially interesting for us is of course middleware. But because of the complexity of existing middlewares, user studies are very difficult to design, since it can take hours to even understand basic usage examples, by far exceeding the time that is normally allocated per user in a study. Therefore, a way needs to be found for executing a study where users can solve middleware problems within a short timeframe, while still getting useful results. Possible ideas are to simplify the APIs for the study, or to provide parts of the solution already finished to the user, but it needs to be checked whether the results are still useful under such simplified conditions. Also, the approach presented in [85] could be evaluated, where a complex API was analyzed with multiple users in a kind of workshop, instead of a traditional user study.
- When working with an API, often a programmer doesn't create something new, but rather extends and enhances code that already exists. Therefore, a valuable addition to the existing study results would be to conduct usability studies that check the *readability* and/or *maintainability* of existing code that was created using a certain API.
- Finally, we identified some aspects of code completion mechanisms that could be improved. This includes a better ordering of overloads (simple before complex types), and allowing the API to specify the ordering. New code completion mechanisms could be created that take these findings into account, and then again are tested with usability studies.

APPENDIX A

Usability Study Questionnaires

A.1 General Questionnaire Used in Both Usability Studies



Usability Test – Abschließende Fragen

Getestetes API	
Programmiersprache	

Name der Testperson	
Ausbildung/aktuelle Beschäftigung	
Alter	
Jahre Programmiererfahrung ca.	

Zu dem API, mit dem Sie gerade gearbeitet haben, würden wir Sie bitten, folgende Fragen zu beantworten:

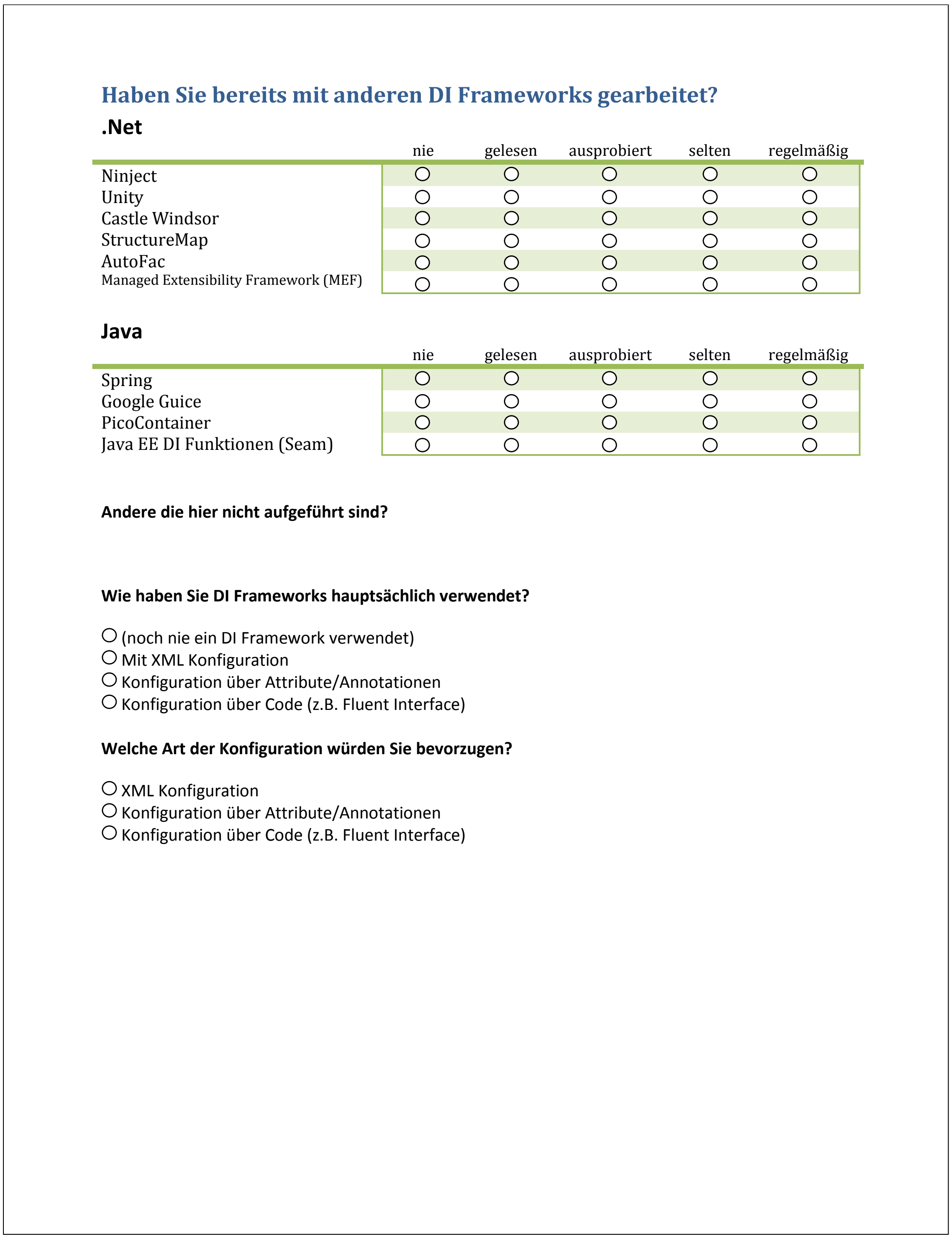
Das API war	Einfach	☐☐☐☐☐	Komplex
Die Verwendung des APIs war	Angenehm	☐☐☐☐☐	Irritierend/Frustrierend
Wie schnell konnten die Aufgaben mit dem API gelöst werden?	Schnell	☐☐☐☐☐	Langsam
Betreffend des Aufgabenbereichs des APIs war meine Vorerfahrung	Hoch	☐☐☐☐☐	Gering

Welche Dinge sind Ihnen am API besonders positiv oder negativ aufgefallen?
(z.B. Klassen, Methoden oder sonstige Eigenschaften)

Gibt es sonst noch etwas, das Sie zum API oder dem Test selbst anmerken möchten?

Vielen Dank für Ihre Teilnahme!

A.2 Additional Questions Concerning DI Frameworks in Study 2



Haben Sie bereits mit anderen DI Frameworks gearbeitet?

.Net

	nie	gelesen	ausprobiert	selten	regelmäßig
Ninject	○	○	○	○	○
Unity	○	○	○	○	○
Castle Windsor	○	○	○	○	○
StructureMap	○	○	○	○	○
AutoFac	○	○	○	○	○
Managed Extensibility Framework (MEF)	○	○	○	○	○

Java

	nie	gelesen	ausprobiert	selten	regelmäßig
Spring	○	○	○	○	○
Google Guice	○	○	○	○	○
PicoContainer	○	○	○	○	○
Java EE DI Funktionen (Seam)	○	○	○	○	○

Andere die hier nicht aufgeführt sind?

Wie haben Sie DI Frameworks hauptsächlich verwendet?

○ (noch nie ein DI Framework verwendet)
○ Mit XML Konfiguration
○ Konfiguration über Attribute/Annotationen
○ Konfiguration über Code (z.B. Fluent Interface)

Welche Art der Konfiguration würden Sie bevorzugen?

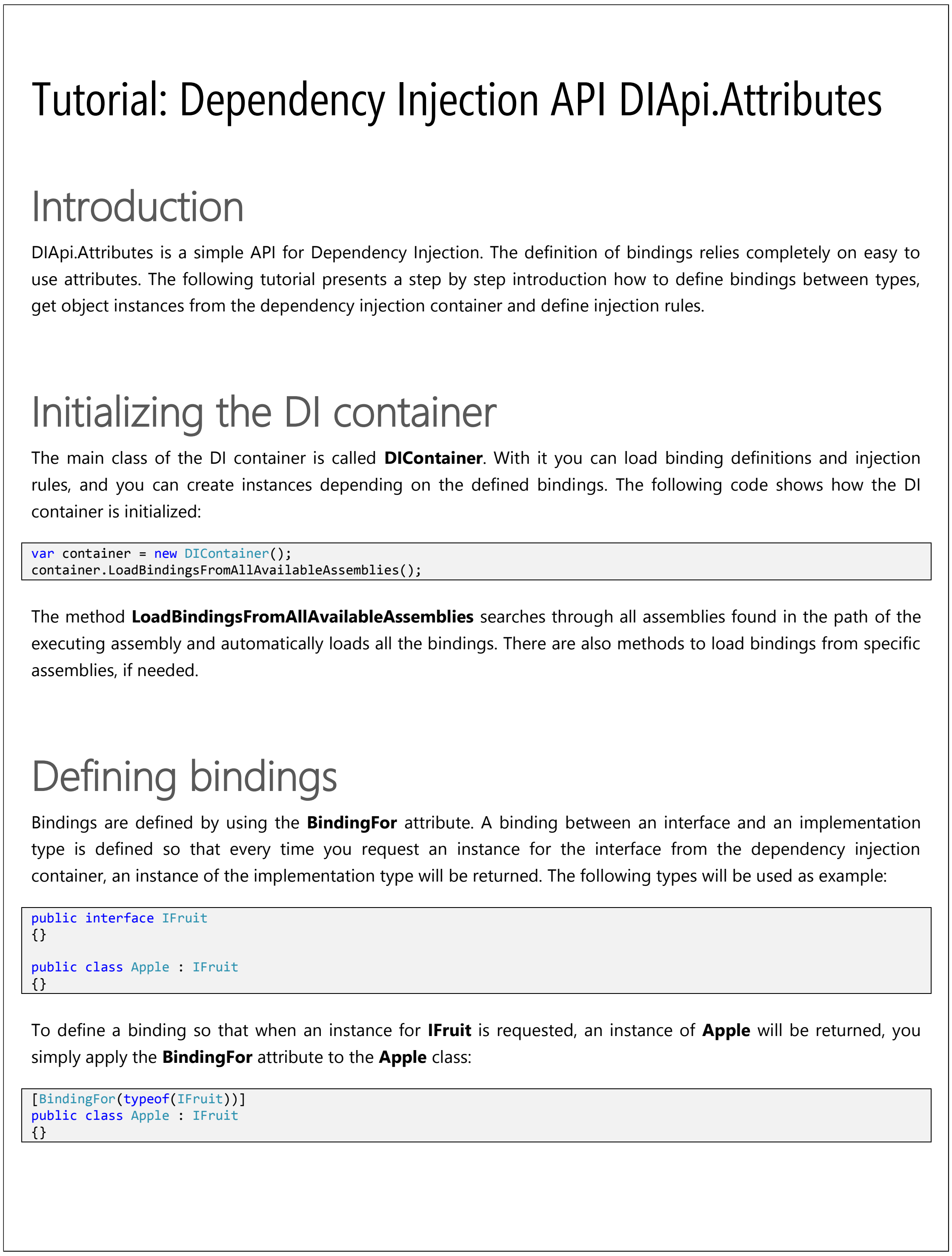
○ XML Konfiguration
○ Konfiguration über Attribute/Annotationen
○ Konfiguration über Code (z.B. Fluent Interface)

APPENDIX B

Tutorials for Usability Study 2

This appendix contains the tutorials that were used in the dependency injection usability study. Only the .Net variants are shown (the Java variants are equivalent, except for the code examples).

B.1 Annotations/Attributes API



Tutorial: Dependency Injection API DIApi.Attributes

Introduction

DIApi.Attributes is a simple API for Dependency Injection. The definition of bindings relies completely on easy to use attributes. The following tutorial presents a step by step introduction how to define bindings between types, get object instances from the dependency injection container and define injection rules.

Initializing the DI container

The main class of the DI container is called **DIContainer**. With it you can load binding definitions and injection rules, and you can create instances depending on the defined bindings. The following code shows how the DI container is initialized:

```
var container = new DIContainer();
container.LoadBindingsFromAllAvailableAssemblies();
```

The method **LoadBindingsFromAllAvailableAssemblies** searches through all assemblies found in the path of the executing assembly and automatically loads all the bindings. There are also methods to load bindings from specific assemblies, if needed.

Defining bindings

Bindings are defined by using the **BindingFor** attribute. A binding between an interface and an implementation type is defined so that every time you request an instance for the interface from the dependency injection container, an instance of the implementation type will be returned. The following types will be used as example:

```
public interface IFruit
{}

public class Apple : IFruit
{}
```

To define a binding so that when an instance for **IFruit** is requested, an instance of **Apple** will be returned, you simply apply the **BindingFor** attribute to the **Apple** class:

```
[BindingFor(typeof(IFruit))]
public class Apple : IFruit
{}
```

Additional binding options

- **Named bindings**: You can define multiple bindings for the same interface. To distinguish these bindings, you can give each binding a name. You can then get instances from the container by using this name.
- **Scope of a binding**: With scope you can define when a new instance is created. There are three different scopes available:
    - *Transient (Default)*: A new instance is created every time it is requested from the container.
    - *Singleton*: Only a single instance is created, the container always returns the same instance.
    - *Thread*: One instance per thread is created

```
[BindingFor(typeof(IFruit), Name = "redfruit", Scope = Scope.Singleton)]
class Strawberry : IFruit
{}
```

Getting instances from the DI container

With the **DIContainer** class, you can now request instances. If bindings like above are defined, you can request instances by type and optionally name, using the container's **Get** method:

```
var fruit1 = container.Get<IFruit>(); //returns an instance of Apple
var fruit2 = container.Get<IFruit>("redfruit"); //returns an instance of Strawberry
```

Constructor Injection

Constructor Injection is one of the most common usage scenarios of DI containers. It means that when an instance is requested from the container, it automatically resolves all constructor parameters and fills in appropriate values.

For example, we want to instantiate this class:

```
public class Dessert
{
          public IFruit Fruit { get; private set; }
          public ICake Cake { get; private set; }

          public Dessert(IFruit fruit, ICake cake)
          {
                    Fruit = fruit;
                    Cake = cake;
          }
}

...
var dessert = container.Get<Dessert>(); //automatically injects IFruit and ICake if there are appropriate bindings
```

If the container knows bindings for **IFruit** and **ICake**, it fills in the appropriate instances automatically.

Configuring what is injected

You can configure different aspects of the injection process:

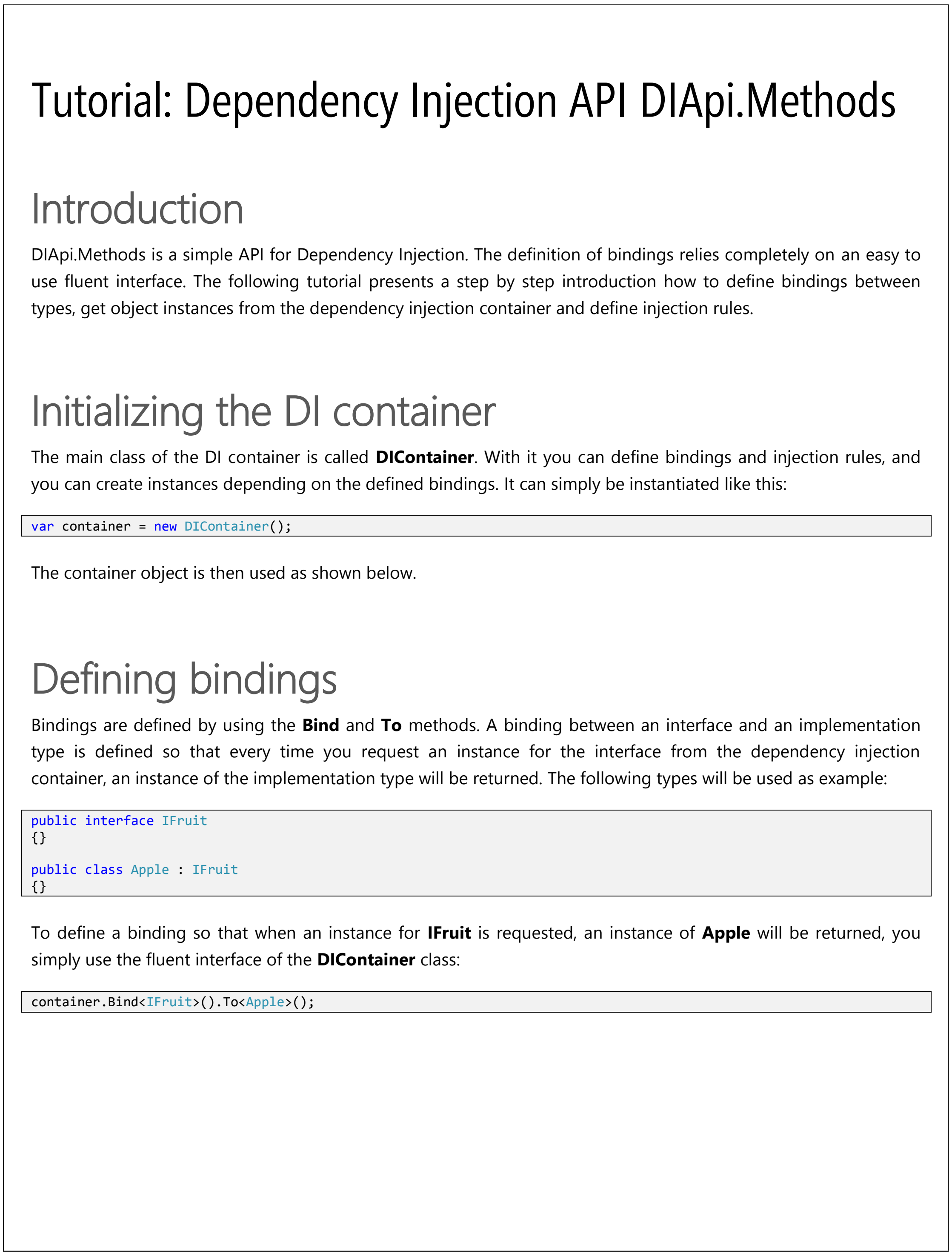
- Which **constructor** is used (by default, the constructor with the most parameters will be used)
- Which bindings are used for the **constructor parameters** (e.g. if there are multiple named bindings for the same type, you can define that a certain one is injected)
- If any public **properties** of the class should be injected

This can all be done using the **Inject** attribute. The following example shows how it can be used:

```
[BindingFor(typeof(IFruit), Name = "redfruit")]
class Strawberry : IFruit { }

[BindingFor(typeof(ICake))]
class CheeseCake : ICake { }

public class Dessert
{
          public IFruit Fruit { get; set; }

          [Inject] //tells the DI container to inject an instance of ICake into this property
          public ICake Cake { get; set; }

          public Dessert (IFruit fruit, ICake cake)
          {
                    Fruit = fruit;
                    Cake = cake;
          }

          [Inject] //tells the DI container to use this constructor
          public Dessert(
                    [Inject("redfruit")] IFruit fruit) //tells the DI container to use the "redfruit" binding
                    : this(fruit, null)
          {
          }
}
```

As the example shows you can use the Inject attribute for constructors, constructor parameters and properties. You can optionally specify the name of the binding, if a specific named binding should be used for injection.

B.2 Fluent Interface API



Tutorial: Dependency Injection API DIApi.Methods

Introduction

DIApi.Methods is a simple API for Dependency Injection. The definition of bindings relies completely on an easy to use fluent interface. The following tutorial presents a step by step introduction how to define bindings between types, get object instances from the dependency injection container and define injection rules.

Initializing the DI container

The main class of the DI container is called **DIContainer**. With it you can define bindings and injection rules, and you can create instances depending on the defined bindings. It can simply be instantiated like this:

```
var container = new DIContainer();
```

The container object is then used as shown below.

Defining bindings

Bindings are defined by using the **Bind** and **To** methods. A binding between an interface and an implementation type is defined so that every time you request an instance for the interface from the dependency injection container, an instance of the implementation type will be returned. The following types will be used as example:

```
public interface IFruit
{}

public class Apple : IFruit
{}
```

To define a binding so that when an instance for **IFruit** is requested, an instance of **Apple** will be returned, you simply use the fluent interface of the **DIContainer** class:

```
container.Bind<IFruit>().To<Apple>();
```

Additional binding options

- **Named bindings**: You can define multiple bindings for the same interface. To distinguish these bindings, you can give each binding a name. You can then get instances from the container by using this name.
- **Scope of a binding**: With scope you can define when a new instance is created. There are three different scopes available:
    - *Transient (Default)*: A new instance is created every time it is requested from the container.
    - *Singleton*: Only a single instance is created, the container always returns the same instance.
    - *Thread*: One instance per thread is created

```
container.Bind<IFruit>().To<Strawberry>().Named("redfruit").InSingletonScope();
```

Getting instances from the DI container

With the **DIContainer** class, you can now request instances. If bindings like above are defined, you can request instances by type and optionally name, using the container's **Get** method:

```
var fruit1 = container.Get<IFruit>(); //returns an instance of Apple
var fruit2 = container.Get<IFruit>("redfruit"); //returns an instance of Strawberry
```

Constructor Injection

Constructor Injection is one of the most common usage scenarios of DI containers. It means that when an instance is requested from the container, it automatically resolves all constructor parameters and fills in appropriate values.

For example, we want to instantiate this class:

```
public class Dessert
{
          public IFruit Fruit { get; private set; }
          public ICake Cake { get; private set; }

          public Dessert(IFruit fruit, ICake cake)
          {
                    Fruit = fruit;
                    Cake = cake;
          }
}

...
var dessert = container.Get<Dessert>(); //automatically injects IFruit and ICake if there are appropriate bindings
```

If the container knows bindings for **IFruit** and **ICake**, it fills in the appropriate instances automatically.

Configuring what is injected

You can configure different aspects of the injection process:

- Which **constructor** is used (by default, the constructor with the most parameters will be used)
- Which bindings are used for the **constructor parameters** (e.g. if there are multiple named bindings for the same type, you can define that a certain one is injected)
- If any public **properties** of the class should be injected

This can all be done using the fluent interface. The following example shows a class for which special injection rules should be defined:

```
public class Dessert
{
	public IFruit Fruit { get; set; }

	//an instance of ICake should be injected into this property
	public ICake Cake { get; set; }

	public Dessert(IFruit fruit, ICake cake)
	{
		Fruit = fruit;
		Cake = cake;
	}

	//this constructor should be used for instantiation
	public Dessert(IFruit fruit) //the binding "redfruit" should be used for the fruit parameter
		: this(fruit, null)
	{
	}
}
```

For the requirements stated in the code above, we can use the fluent interface as follows:

```
container.Bind<IFruit>().To<Strawberry>().Named("redfruit");
container.Bind<ICake>().To<CheeseCake>();

//to define which constructor to use:
container.WhenInstantiating<Dessert>().UseConstructorWithTypes<IFruit>();

//to define that a named binding should be injected into a constructor parameter:
//(if the default binding is used, defining this is not necessary)
container.WhenInstantiating<Dessert>().ForConstructorParam("fruit").Inject("redfruit");

//to define that a property should be injected:
container.WhenInstantiating<Dessert>().ForProperty("Cake").InjectDefault();

//the methods can also be linked together like this:
container.WhenInstantiating<Dessert>()
	.UseConstructorWithTypes<IFruit>()
	.ForConstructorParam("fruit").Inject("redfruit")
	.ForProperty("Cake").InjectDefault();
```

As the example shows, the fluent interface can be used to easily define injection rules for constructors, constructor parameters and properties.

B.3 XML API

Tutorial: Dependency Injection API DIApi.Xml

Introduction

DIApi.Xml is a simple API for Dependency Injection. The definition of bindings relies completely on xml configuration. The following tutorial presents a step by step introduction how to define bindings between types, get object instances from the dependency injection container and define injection rules.

Initializing the DI container

The main class of the DI container is called **DIContainer**. With it you can define bindings and injection rules, and you can create instances depending on the defined bindings. It can simply be instantiated like this:

```
var container = new DIContainer("config.xml");
```

The file config.xml must be an xml file containing the binding configuration for the DI container. How this file looks like can be seen below.

Defining bindings

Bindings are defined by using the xml configuration file. A binding between an interface and an implementation type is defined so that every time you request an instance for the interface from the dependency injection container, an instance of the implementation type will be returned. The following types will be used as example:

```
namespace MyClasses
{
          public interface IFruit
          {}

          public class Apple : IFruit
          {}
}
```

To define a binding so that when an instance for **IFruit** is requested, an instance of **Apple** will be returned, you simply use the following xml configuration:

```
<?xml version="1.0" encoding="utf-8" ?>
<DIContainer>
  <Bindings>
    <Binding type="MyClasses.IFruit" bindTo="MyClasses.Apple"/>
  </Bindings>
</DIContainer>
```

Additional binding options

- **Named bindings**: You can define multiple bindings for the same interface. To distinguish these bindings, you can give each binding a name. You can then get instances from the container by using this name.
- **Scope of a binding**: With scope you can define when a new instance is created. There are three different scopes available:
    - *Transient (Default)*: A new instance is created every time it is requested from the container.
    - *Singleton*: Only a single instance is created, the container always returns the same instance.
    - *Thread*: One instance per thread is created

```
<Binding type="MyClasses.IFruit" bindTo="MyClasses.Strawberry" name="redfruit" scope="Singleton"/>
```

Getting instances from the DI container

With the **DIContainer** class, you can now request instances. If bindings like above are defined, you can request instances by type and optionally name, using the container's **Get** method:

```
var fruit1 = container.Get<IFruit>(); //returns an instance of Apple
var fruit2 = container.Get<IFruit>("redfruit"); //returns an instance of Strawberry
```

Constructor Injection

Constructor Injection is one of the most common usage scenarios of DI containers. It means that when an instance is requested from the container, it automatically resolves all constructor parameters and fills in appropriate values.

For example, we want to instantiate this class:

```
public class Dessert
{
          public IFruit Fruit { get; private set; }
          public ICake Cake { get; private set; }

          public Dessert(IFruit fruit, ICake cake)
          {
                    Fruit = fruit;
                    Cake = cake;
          }
}

...
var dessert = container.Get<Dessert>(); //automatically injects IFruit and ICake if there are appropriate bindings
```

If the container knows bindings for **IFruit** and **ICake**, it fills in the appropriate instances automatically.

Configuring what is injected

You can configure different aspects of the injection process:

- Which **constructor** is used (by default, the constructor with the most parameters will be used)
- Which bindings are used for the **constructor parameters** (e.g. if there are multiple named bindings for the same type, you can define that a certain one is injected)
- If any public **properties** of the class should be injected

This can all be done using the fluent interface. The following example shows a class for which special injection rules should be defined:

```
public class Dessert
{
        public IFruit Fruit { get; set; }

        //an instance of ICake should be injected into this property
        public ICake Cake { get; set; }

        public Dessert(IFruit fruit, ICake cake)
        {
                Fruit = fruit;
                Cake = cake;
        }

        //this constructor should be used for instantiation
        public Dessert(IFruit fruit) //the binding "redfruit" should be used for the fruit parameter
                : this(fruit, null)
        {
        }
}
```

For the requirements stated in the code above, we can use the fluent interface as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<DIContainer>
  <Bindings>
    <Binding type="MyClasses.IFruit" bindTo="MyClasses.Strawberry" name="redfruit"/>
    <Binding type="MyClasses.ICake" bindTo="MyClasses.CheeseCake"/>
  </Bindings>
  <Instantiations>
    <Instantiation type="MyClasses.Dessert">
      <Constructor> <!-- the constructor is chosen according to this list of params -->
        <Param name ="fruit" binding="redfruit"/> <!-- the optional binding attribute allows defining that a
                                                    specific binding should be used-->
      </Constructor>
      <Properties> <!-- all properties listed here will be injected -->
        <Property name ="Cake"/>
      </Properties>
    </Instantiation>
  </Instantiations>
</DIContainer>
```

As the example shows, the xml configuration can be used to easily define injection rules for constructors, constructor parameters and properties.

APPENDIX C

R Code for Plots and Statistics

C.1 Usability Study 1

Reading and preparing the data set for statistical evaluation:

```
#convert mm:ss strings into seconds values
time.to.seconds <- function(time) {
   time <- strsplit(time, ":")[[1]]
   return((as.numeric(time[1]) * 60) + (as.numeric(time[2]))   )
 }

#read data from csv file
path <- "... path of the csv file ..."
dat <- as.data.frame(read.csv2(path))

#for all time columns convert the mm:ss value into seconds
timecolumns <- c("Overall", "Search.for.Class", "Instantiate.Class",
    "Search.for.Method", "Use.Method", "Search.Inst..Class",
    "Search.Use.Method")
for (colname in timecolumns)
{
   dat[,colname] <-
       sapply(as.character(dat[,colname]),time.to.seconds)
}

#some columns have to be converted into text, or else some qplot
    functions don't work (like side-by-side boxplots)
dat[,"API.Version"] <- as.character(dat[,"API.Version"])
dat[,"Task.ID.Label"] <- as.character(dat[,"Task.ID"])
dat[,"Parameteranzahl.Label"] <-
    as.character(dat[,"Parameteranzahl"])
dat[,"Programming.XP.Label"] <- as.character(dat[,"Programming.XP"])
dat[,"Domain.Knowledge.Label"] <-
    as.character(dat[,"Domain.Knowledge"])
#attach headers
attach(dat)
```

Boxplot comparison plots:

```
#graphics with ggplot2
install.packages("ggplot2")
library(ggplot2)

#### search for class ####
qplot(API.Version, Search.for.Class, data =
    dat[which(Used.exp..Class=="x" & Task.ID == 1),], geom =
    "boxplot", xlab="API Version", ylab="Time to search for class
    (seconds)") + scale_y_continuous(breaks = seq(0, 1000, 20))

#### instantiate class ####
#api 1 vs 2
graphdata <- dat[which(API.Version!="0" & Task.ID < 3),] #tasks 1+2
graphdata <- dat[which(API.Version!="0"),] #all tasks
graphdata <- dat[which(Task.ID < 3 & API.Version==1 &
    Used.exp..Class=="x"),] #tasks 1+2 only factory
qplot(API.Version, Instantiate.Class, data = graphdata, geom =
    "boxplot", xlab="API Version", ylab="Time to instantiate class
    (seconds)") + scale_y_continuous(breaks = seq(0, 1000, 20))
#Factory .Net vs Java
qplot(Language, Search.for.Class+Instantiate.Class, data =
    graphdata, geom = "boxplot", ylab="Search and inst. class with
    factory (seconds)")

#### search for method ####
graphdata <- dat[which(Video.Analysis=="x" & Used.exp..Method=="x" &
    Success=="x" & Task.ID!=3),] #overall
graphdata <- dat[which(Video.Analysis=="x" & Used.exp..Overload=="x"
    & Searched.by.List=="x" & Search.for.Method > 10 & Success=="x" &
    Task.ID!=3),] #search by list, used exp overload, time > 10s
#api 1 vs 2
qplot(API.Version, Search.for.Method, data = graphdata, geom =
    "boxplot", xlab="API version", ylab="Time to search for method
    (seconds)") + scale_y_continuous(breaks = seq(0, 1000, 20))
#Eclipse vs VS vs VS+ReSharper
graphdata <- dat[which(Video.Analysis=="x" & Used.exp..Method=="x" &
    Searched.by.List=="x" & Success=="x" & Task.ID!=3),] #search by
    list, used exp method
graphdata <- dat[which(Video.Analysis=="x" & Searched.by.List=="x" &
    Success=="x" & Task.ID!=3),] #search by list
qplot(IDE, Search.for.Method, data = graphdata, geom = "boxplot",
    ylab="Time to search for method (seconds)") +
    scale_y_continuous(breaks = seq(0, 1000, 20))
#method with many overloads / members with same prefix vs. few
    (addFile vs removeEntry)
qplot(Task.ID.Label, Search.for.Method, data =
    dat[which(Used.exp..Class=="x" & (Task.ID==1 | Task.ID==6)),],
    geom = "boxplot", xlab="Searched Method", ylab="Time to search
    for method (seconds)") +
    scale_x_discrete(labels=c("addFile","removeEntry"))

#### expected vs. not expected ####
#Search for Method - Use exp. Method
qplot(Used.exp..Method, Search.for.Method, data = dat, geom =
    "boxplot", ylab="search for method (seconds)", xlab="used
    expected method?")
#Search for Method - Use exp. Overload
```

```
qplot(Used.exp..Overload, Search.for.Method, data = dat, geom =
    "boxplot", ylab="search for method (seconds)", xlab="used
    expected overload?")
#Seach + Use Method - Use exp. Method
graphdata <- dat[which(Used.exp..Class=="x" & Success == "x"),]
qplot(Used.exp..Method, Search.Use.Method, data = graphdata, geom =
    "boxplot", ylab="Time to search and use method (seconds)",
    xlab="Used expected method?") + scale_x_discrete(limits =
    c("x",""),labels=c("x" = "yes","no")) + scale_y_continuous(breaks
    = seq(0, 1000, 50))
#Seach + Use Method - Use exp. Overload
graphdata <- dat[which(Used.exp..Class=="x" & Success == "x"),]
qplot(Used.exp..Overload, Search.Use.Method, data = graphdata, geom
    = "boxplot", ylab="Time to search and use method (seconds)",
    xlab="Used expected overload?") + scale_x_discrete(limits =
    c("x",""),labels=c("x" = "yes","no")) + scale_y_continuous(breaks
    = seq(0, 1000, 50))

#### Percent used not expected method per programming xp ####
library(scales)
p <- qplot(Programming.XP, data = dat[which(Used.exp..Class=="x" &
    Success == "x"),], geom="bar", fill = Used.exp..Method, position
    = "fill", ylab = "% of cases where not the exp. method was used")
p <- qplot(Programming.XP, data = dat[which(Used.exp..Class=="x" &
    Success == "x"),], geom="bar", fill = Used.exp..Overload,
    position = "fill", ylab = "% of cases where not the exp. overload
    was used")
p + scale_y_continuous(labels = percent, limits=c(0,0.4)) +
    scale_fill_manual("Used exp. Method?", limits = c(""),
    labels=c("x" = "yes","no") , values = c("red","green")) +
    scale_x_discrete("Years of programming experience", limits =
    c(1,2,3,4,5,6,7,8,9,10))+ theme(legend.position = "none")

#### using methods with different parameter counts ####
graphdata <- dat[which(Success=="x" & (Parameteranzahl==1 |
    Parameteranzahl==2 | Parameteranzahl==4)),]
qplot(Parameteranzahl.Label, Use.Method, data = graphdata, geom =
    "boxplot",xlab="Number of parameters",ylab="Time to use method
    (seconds)")

#### task 5 search+use method api 1 vs api 2 (more params vs more
    overloads) ####
qplot(API.Version, Search.Use.Method, data = dat[which(Task.ID==5 &
    Used.exp..Method=="x"),], geom = "boxplot", xlab="API version",
    ylab="Time to search and use method (seconds)") +
    scale_y_continuous(breaks = seq(0, 1000, 50))
```

Learning effect and correlation plots:

```
#### learning effect: search+instantiate class vs task id ####
qplot(Task.ID, Search.Inst..Class, data =
    dat[which(Used.exp..Class=="x"),], geom = c("point",
    "smooth"),xlab="Task number",ylab="Time to search and instantiate
    class (seconds)")

#### correlation of performance to years of programming xp ####
#overall values from every task
```

```
qplot(Programming.XP, Overall, data = dat[which(Success=="x"),],
    geom = c("point", "smooth"), xlab="Programming experience
    (years)",ylab="Time to solve task  (seconds)")

#### correlation of performance to domain knowledge ####
#overall values from every task
qplot(Domain.Knowledge, Overall, data = dat[which(Success=="x"),],
    geom = c("point", "smooth"), xlab="Domain knowledge (1 - 5,
    5=highest)",ylab="Time to solve task  (seconds)")
```

Questionnaire plots and success rate correlations:

```
#simplicity
qplot(API.Version, Q1_Simplicity, data = dat2, geom = "boxplot",
    xlab="API version", ylab="Simplicity") +
    scale_y_continuous(limits=c(2.0,5.0))
#satisfaction
qplot(API.Version, Q2_Satisfaction, data = dat2, geom = "boxplot",
    xlab="API version", ylab="Satisfaction") +
    scale_y_continuous(limits=c(2.0,5.0))
#speed
qplot(API.Version, Q3_Speed, data = dat2, geom = "boxplot",
    xlab="API version", ylab="Speed") +
    scale_y_continuous(limits=c(2.0,5.0))

#plots: num task vs xp / domain knowledge
#uses small jitter so that all points become visible
library(ggplot2)
qplot(Years.XP, jitter(Num.Successful.Tasks, amount=0.15), data =
    dat, geom = c("point", "smooth"), xlab="Programming experience
    (years)",ylab="Number of successful tasks")
qplot(Q4_Domain_Knowledge, jitter(Num.Successful.Tasks,
    amount=0.15), data = dat[which(Success=="x"),], geom = c("point",
    "smooth"), xlab="Domain knowledge (1-5, 5=highest)",ylab="Number
    of successful tasks")
```

Statistics (calculation of p-values):

```
#wilcoxon rank sum test (take any definition of x and y according to
    the plots)
wilcox.test(x, y)
wilcox.test(x, y, alternative="greater")
wilcox.test(x, y, alternative="less")
median(x)
median(y)

#correlation tests
cor.test(Programming.XP, Overall, method = "pearson", alternative =
    "less")
cor.test(Domain.Knowledge, Overall, method = "pearson", alternative
    = "less")
```

C.2 Usability Study 2

Reading and preparing the data set for statistical evaluation, the calculation of p-values, as well as the evaluation of the questionnaire, was done in the same way as for study 1.

Boxplot comparison plots:

```
### task times ###
#comparison of overall times per task and API in a single plot
qplot(Task.ID.Label, Time.Overall, data = dat[which(Finished=="x" &
    Final.API=="x"),], fill=API.Version, geom = "boxplot", xlab="Task
    Number", ylab="Overall Time (seconds)") +
    scale_fill_grey(name="API Version", start=0.35,end=0.95)
#simple tasks - bind ConsoleLogger incl. tutorial
qplot(API.Version, Time.Bind.ConsoleLogger + Time.Tutorial.Bind,
    data = dat[which(Finished=="x"),], geom = "boxplot", xlab="API
    version", ylab="Time for Binding + Tutorial (seconds)")
#complex tasks - inject service inkl. tutorial
qplot(API.Version, Time.Tutorial.Injection + Time.Injection.Service,
    data = dat[which(Finished=="x" & Final.API=="x"),], geom =
    "boxplot", xlab="API version", ylab="Time for Injection +
    Tutorial (seconds)")

### code switches ###
#overall
qplot(API.Version, Switches.Overall, data = dat[which(Finished=="x"
    & Eye.Tracking=="x"),], geom = "boxplot", xlab="API version",
    ylab="Number of switches between code files")
#for simple tasks
qplot(API.Version, Switches.Bind.ConsoleLogger, data =
    dat[which(Finished=="x" & Eye.Tracking=="x"),], geom = "boxplot",
    xlab="API version", ylab="Number of Switches for Binding")
#for complex tasks
qplot(API.Version, Switches.Injection.Service, data =
    dat[which(Finished=="x" & Eye.Tracking=="x"),], geom = "boxplot",
    xlab="API version", ylab="Number of Switches for Injection")

### tutorial times ###
#time spent in tutorial
qplot(API.Version, Time.Tutorial.Overall, data =
    dat[which(Finished=="x" & Task.ID != 6),], geom = "boxplot",
    xlab="API version", ylab="Tutorial Time (seconds)", ylim=c(0,250))
#times switched to tutorial
qplot(API.Version, Times.Switched.to.Tutorial, data =
    dat[which(Finished=="x" & Final.API=="x" & Task.ID != 6),], geom
    = "boxplot", xlab="API version", ylab="Number of tutorial
    switches")
#time spent looking at headers and code examples
qplot(API.Version,
    Time.Tutorial.Overall*(Percent.Tutorial.Code+0.5*(1-
    Percent.Tutorial.Code)),data = dat[which(Finished=="x" &
    Eye.Tracking=="x" & Task.ID != 6),], geom = "boxplot", xlab="API
    version", ylab="Tutorial time reading only code and headers
    (seconds)", ylim=c(0,200))
#time spent NOT looking at headers and code examples (= reading text)
qplot(API.Version, Time.Tutorial.Overall-(Time.Tutorial.Overall*
    (Percent.Tutorial.Code+0.5*(1-Percent.Tutorial.Code))),data =
    dat[which(Finished=="x" & Eye.Tracking=="x" & Task.ID != 6),],
    geom = "boxplot", xlab="API version", ylab="Tutorial time reading
    text (seconds)", ylim=c(0,100))
#percent of time reading only header and code examples
qplot(API.Version,
    Percent.Tutorial.Code*100+50*(1-Percent.Tutorial.Code),data =
    dat[which(Finished=="x" & Eye.Tracking=="x" &
    Time.Tutorial.Overall > 0),], geom = "boxplot", xlab="API
    version", ylab="% of time reading headers and code examples")
```

Comparison of search places (TU/CC/NO):

```
library('scales')
dat$Search.Bind.ConsoleLogger <- factor
    (dat$Search.Bind.ConsoleLogger, levels = c("TU", "CC", "NO"))
dat$Search.Injection.Service <- factor(dat$Search.Injection.Service,
    levels = c("TU", "CC", "NO"))
#bind consolelogger over all tasks
qplot(API.Version, fill=Search.Bind.ConsoleLogger, data =
    dat[which(Finished=="x"),], geom="bar", position="fill",
    xlab="API version", ylab="") + scale_y_continuous(label=percent)
    + scale_fill_grey(name="Source (Binding)", start=0.35,end=0.95,
    labels = c("TU"="Tutorial","CC"="Auto Compl.","NO"="None"))
#inject service over all tasks
qplot(API.Version, fill=Search.Injection.Service, data =
    dat[which(Finished=="x" & (Task.ID == 3 | Task.ID == 4 |
    Task.ID==6)),], geom="bar", position="fill", xlab="API version",
    ylab="") + scale_y_continuous(label=percent) +
    scale_fill_grey(name="Source (Injection)", start=0.35,end=0.95,
    labels = c("TU"="Tutorial","CC"="Auto Compl.","NO"="None"))
```

User experience correlation plots:

```
### correlation of time to years of programming experience ###
qplot(XP.Years, Time.Overall, data = dat[which(Finished=="x" &
    Task.ID<6),], geom = c("point"), xlab="Programming experience
    (years)",ylab="Time to solve task  (seconds)") + facet_grid(. ~
    API.Version) + scale_y_continuous(breaks = seq(0, 1000, 200)) +
    geom_smooth(method="lm")

### correlation of time to domain knowledge ###
qplot(Q.Domain.Knowledge, Time.Overall, data =
    dat[which(Finished=="x" & Task.ID<6),], geom = c("point"),
    xlab="Domain knowledge (1 - 5, 5=highest)",ylab="Time to solve
    task  (seconds)") + facet_grid(. ~ API.Version) +
    scale_y_continuous(breaks = seq(0, 1000, 200)) +
    geom_smooth(method="lm")

### correlation of time to DI framework experience ###
qplot(XP.Frameworks.Overall, Time.Overall, data =
    dat[which(Finished=="x" & Task.ID<6),], geom = c("point"),
    xlab="DI framework experience (higher is more
    experienced)",ylab="Time to solve task  (seconds)") +
    facet_grid(. ~ API.Version) + scale_y_continuous(breaks = seq(0,
    1000, 200)) + geom_smooth(method="lm")
```

APPENDIX D

Listings and Tables for Middleware Usability Evaluation

This appendix contains all listings and evaluation tables for the middleware usability evaluation presented in chapter 8.

D.1 WCF

D.1.1 Simple Message Transfer

Contract:

```
[ServiceContract]
public interface IService {
   [OperationContract]
   void Ping(string text);
}
```

Server:

```
//service
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class MyService : IService {
   public void Ping(string text) {
      Console.WriteLine("received: " + text);
   }
}

//main method
using (var svh = new ServiceHost(typeof (MyService), new
    Uri("net.tcp://localhost:9001")))
{
   svh.AddServiceEndpoint(typeof (IService), new NetTcpBinding(),
       "service");
   svh.Open();
   Console.ReadLine();
}
```

Client:

```
using (var cf = new ChannelFactory<IService>(new NetTcpBinding(),
    "net.tcp://localhost:9001"))
{
   IService s = cf.CreateChannel();
   s.Ping("hello, world!");
   Console.ReadLine();
}
```

D.1.2 Request with Multiple Responses

Contract:

```
[ServiceContract(CallbackContract =
    typeof(IProgressNotificationSink))]
public interface IService {
   [OperationContract]
   string Ping(string text);
}

public interface IProgressNotificationSink {
   [OperationContract(IsOneWay = true)]
   void Progress(int progress);
}
```

Server:

```
//service
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall,
    ConcurrencyMode = ConcurrencyMode.Reentrant)]
class MyService : IService {
   public string Ping(string text) {
      Console.WriteLine("processing: {0}", text);
      var pns = OperationContext.Current
          .GetCallbackChannel<IProgressNotificationSink>();
      return WorkProcessor.Process(text, pns.Progress);
   }
}

//main method
using (var svh = new ServiceHost(typeof (MyService), new
    Uri("net.tcp://localhost:9001")))
{
   svh.AddServiceEndpoint(typeof (IService), new NetTcpBinding(),
       "service");
   svh.Open();

   Console.ReadLine();
}
```

Client:

```
//implementation of callback channel interface
class MyProgressNotificationSink : IProgressNotificationSink {
   public void Progress(int progress) {
      Console.WriteLine("  progress: {0}", progress);
   }
}

//main method
IProgressNotificationSink pns = new MyProgressNotificationSink();
var ctx = new InstanceContext(pns);
using (var cf = new DuplexChannelFactory<IService>(ctx, new
    NetTcpBinding(), "net.tcp://localhost:9001/service"))
{
   IService s = cf.CreateChannel();
   Console.WriteLine("received: {0}", s.Ping("hello, world!"));
}
```

D.1.3 Publish/Subscribe

Contract:

```
[ServiceContract(CallbackContract = typeof(INotificationSink))]
public interface IPubSubService
{
   [OperationContract]
   void Subscribe(string clientID);

   [OperationContract]
   void Unsubscribe(string clientID);

   [OperationContract]
   void Notify(string msg);
}

public interface INotificationSink
{
   [OperationContract(IsOneWay = true)]
   void NotificationReceived(string msg);
}
```

Server:

```
//service
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
    ConcurrencyMode = ConcurrencyMode.Reentrant)]
class PubSubService : IPubSubService
{
   private readonly Dictionary<string, INotificationSink>
       registrations = new Dictionary<string, INotificationSink>();

   public void Subscribe(string clientID)
   {
      var callback = OperationContext.Current
          .GetCallbackChannel<INotificationSink>();
```

```
        registrations[clientID] = callback;
    }

    public void Unsubscribe(string clientID)
    {
        registrations.Remove(clientID);
    }

    public void Notify(string msg)
    {
        var result = WorkProcessor.Process(msg);
        var toBeRemoved = new List<string>();
        Console.WriteLine("sending notifications");
        foreach (var reg in registrations)
        {
            try
            {
                reg.Value.NotificationReceived(result);
            }
            catch (Exception)
            {
                toBeRemoved.Add(reg.Key);
            }
        }
        foreach (var addr in toBeRemoved)
        {
            registrations.Remove(addr);
        }
    }
}

//main method
using (var svh = new ServiceHost(typeof(PubSubService), new
    Uri("net.tcp://localhost:9001")))
{
    svh.AddServiceEndpoint(typeof(IPubSubService), new
        NetTcpBinding(), "service");
    svh.Open();

    Console.ReadLine();
}
```

Client:

```
//notification callback service
class MyNotificationSink : INotificationSink
{
    public void NotificationReceived(string msg)
    {
        Console.WriteLine(" Notification received: {0}", msg);
    }
}

//main method
INotificationSink pns = new MyNotificationSink();
var ctx = new InstanceContext(pns);
using (var cf = new DuplexChannelFactory<IPubSubService>(ctx, new
    NetTcpBinding(), "net.tcp://localhost:9001/service"))
```

```
{
   IPubSubService s = cf.CreateChannel();

   s.Subscribe(MyClientId);
   Console.WriteLine("registered for notifications");
   Console.ReadLine();

   s.Unsubscribe(MyClientId);
   Console.WriteLine("unregistered from notifications");
   Console.ReadLine();
}
```

D.1.4 Usability Evaluation Tables

Table D.1: Evaluation of WCF: Simple message transfer

Search	Usage	Sum	Concept	Usages	Description
30.0	29.0	59.0	ClassUsage	1	ServiceContractAttribute
30.0	29.0	59.0	ClassUsage	1	OperationContractAttribute
30.0	29.0	59.0	ClassUsage	1	ServiceBehaviorAttribute
30.0	21.0	51.0	ClassUsage	1	ServiceHost
30.0	29.4	59.4	ClassUsage	2	NetTcpBinding
30.0	21.0	51.0	ClassUsage	1	ChannelFactory<>
0.0	4.0	4.0	Instantiation	1	[ServiceContractAttribute()]
0.0	4.0	4.0	Instantiation	1	[OperationContractAttribute()]
0.0	4.0	4.0	Instantiation	1	[ServiceBehaviorAttribute()]
11.6	7.0	18.6	FieldAccess	1	ServiceBehaviorAttribute.InstanceContextMode
1.0	19.5	20.5	Instantiation	1	new ServiceHost(Type,Uri[]) InformationLookup Pattern: 0 / 8
21.1	20.0	41.1	MethodCall	1	ServiceHost.AddServiceEndpoint(String,Binding, String) (return value not used)
13.6	5.0	18.6	MethodCall	1	ServiceHost.Open()
0.0	5.6	5.6	Instantiation	2	new NetTcpBinding()
5.0	19.5	24.5	Instantiation	1	new ChannelFactory<>(Binding,String) InformationLookup Pattern: 0 / 8
11.1	13.0	24.1	MethodCall	1	ChannelFactory<>.CreateChannel()
		0.0			InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.
		503.2	**Overall**		

Table D.2: Evaluation of WCF: Request with multiple responses

Search	Usage	Sum	Concept	Usages	Description
30.0	29.0	59.0	ClassUsage	1	ServiceContractAttribute
30.0	40.6	70.6	ClassUsage	2	OperationContractAttribute
30.0	29.0	59.0	ClassUsage	1	ServiceBehaviorAttribute
30.0	21.0	51.0	ClassUsage	1	ServiceHost
30.0	29.4	59.4	ClassUsage	2	NetTcpBinding
30.0	21.0	51.0	ClassUsage	1	DuplexChannelFactory<>
30.0	21.0	51.0	ClassUsage	1	OperationContext
30.0	21.0	51.0	ClassUsage	1	InstanceContext
0.0	4.0	4.0	Instantiation	1	[ServiceContractAttribute()]
10.6	7.0	17.6	FieldAccess	1	ServiceContractAttribute.CallbackContract
0.0	5.6	5.6	Instantiation	2	[OperationContractAttribute()]
10.7	7.0	17.7	FieldAccess	1	OperationContractAttribute.IsOneWay
0.0	4.0	4.0	Instantiation	1	[ServiceBehaviorAttribute()]
11.6	7.0	18.6	FieldAccess	1	ServiceBehaviorAttribute.InstanceContextMode
11.6	7.0	18.6	FieldAccess	1	ServiceBehaviorAttribute.ConcurrencyMode
1.0	19.5	20.5	Instantiation	1	new ServiceHost(Type,Uri[]) InformationLookup Pattern: 0 / 8
21.1	20.0	41.1	MethodCall	1	ServiceHost.AddServiceEndpoint(String,Binding, String) (return value not used)
13.6	5.0	18.6	MethodCall	1	ServiceHost.Open()
1.0	6.5	7.5	Instantiation	1	new InstanceContext(ServiceHostBase)
0.0	5.6	5.6	Instantiation	2	new NetTcpBinding()
13.0	27.0	40.0	Instantiation	1	new DuplexChannelFactory<>(InstanceContext, Binding,String) InformationLookup Pattern: 0 / 8
17.1	13.0	30.1	MethodCall	1	DuplexChannelFactory<>.CreateChannel()
10.0	7.0	17.0	FieldAccess	1	OperationContext.Current
12.1	15.5	27.6	MethodCall	1	OperationContext.GetCallbackChannel<>()
		0.0			InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.
		745.9	**Overall**		

Table D.3: Evaluation of WCF: Publish/subscribe

Search	Usage	Sum	Concept	Usages	Description
30.0	29.0	59.0	ClassUsage	1	ServiceContractAttribute
30.0	63.8	93.8	ClassUsage	4	OperationContractAttribute
30.0	29.0	59.0	ClassUsage	1	ServiceBehaviorAttribute
30.0	21.0	51.0	ClassUsage	1	ServiceHost
30.0	29.4	59.4	ClassUsage	2	NetTcpBinding
30.0	21.0	51.0	ClassUsage	1	DuplexChannelFactory<>
30.0	21.0	51.0	ClassUsage	1	OperationContext
30.0	21.0	51.0	ClassUsage	1	InstanceContext
0.0	4.0	4.0	Instantiation	1	[ServiceContractAttribute()]
10.6	7.0	17.6	FieldAccess	1	ServiceContractAttribute.CallbackContract
0.0	8.8	8.8	Instantiation	4	[OperationContractAttribute()]
10.7	7.0	17.7	FieldAccess	1	OperationContractAttribute.IsOneWay
0.0	4.0	4.0	Instantiation	1	[ServiceBehaviorAttribute()]
11.6	7.0	18.6	FieldAccess	1	ServiceBehaviorAttribute.InstanceContextMode
11.6	7.0	18.6	FieldAccess	1	ServiceBehaviorAttribute.ConcurrencyMode
1.0	19.5	20.5	Instantiation	1	new ServiceHost(Type,Uri[]) InformationLookup Pattern: 0 / 8
21.1	20.0	41.1	MethodCall	1	ServiceHost.AddServiceEndpoint(String,Binding, String) (return value not used)
13.6	5.0	18.6	MethodCall	1	ServiceHost.Open()
1.0	6.5	7.5	Instantiation	1	new InstanceContext(ServiceHostBase)
0.0	5.6	5.6	Instantiation	2	new NetTcpBinding()
13.0	27.0	40.0	Instantiation	1	new DuplexChannelFactory<>(InstanceContext, Binding,String) InformationLookup Pattern: 0 / 8
17.1	13.0	30.1	MethodCall	1	DuplexChannelFactory<>.CreateChannel()
10.0	7.0	17.0	FieldAccess	1	OperationContext.Current
12.1	15.5	27.6	MethodCall	1	OperationContext.GetCallbackChannel<>()
		0.0			InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.
		772.3	**Overall**		

D.2 NServiceBus

D.2.1 Simple Message Transfer

Contract:

```
public class Message : IMessage
{
   public string Msg { get; set; }
}
```

Server:

```
//endpoint configuration
public class EndpointConfig : IConfigureThisEndpoint, AsA_Server
{ }

//handler for incoming messages
public class MsgHandler : IHandleMessages<Message>
{
   public void Handle(Message message)
   {
      Console.WriteLine("Message received: " + message.Msg);
   }
}
```

Client:

```
//endpoint configuration
public class EndpointConfig : IConfigureThisEndpoint, AsA_Client
{ }

//client that runs at startup
public class Client : IWantToRunWhenBusStartsAndStops
{
   public IBus Bus { get; set; }

   public void Start()
   {
      Bus.Send("MsgTransfer.Server", new Message {Msg = "Hello,
         World!"});
   }

   public void Stop()
   {
   }
}
```

D.2.2 Request with Multiple Responses

Contract:

```
public class Request : IMessage
{
   public string Value { get; set; }
}

public class Progress : IMessage
{
   public int Value { get; set; }
}

public class Response : IMessage
{
   public string Value { get; set; }
}
```

Server:

```
//endpoint configuration
public class EndpointConfig : IConfigureThisEndpoint, AsA_Server
{ }

//handler for incoming messages
public class MsgHandler : IHandleMessages<Request>
{
   public IBus Bus { get; set; }

   public void Handle(Request message)
   {
      Console.WriteLine("Message received: " + message.Value);
      var result = WorkProcessor.Process(message.Value, p =>
         Bus.Reply(new Progress {Value = p}));
      Bus.Reply(new Response{Value = result});
   }
}
```

Client:

```
//endpoint configuration
public class EndpointConfig : IConfigureThisEndpoint, AsA_Client
{ }

//client that runs at startup and handles responses
public class Client : IWantToRunWhenBusStartsAndStops,
    IHandleMessages<Progress>, IHandleMessages<Response>
{
   public IBus Bus { get; set; }

   public void Start()
   {
      Bus.Send("MultiResp.Server", new Request {Value = "test123"});
   }

   public void Stop()
   {
   }

   public void Handle(Progress message)
   {
      Console.WriteLine("Progress: " + message.Value);
   }
```

```
    public void Handle(Response message)
    {
        Console.WriteLine("Response: " + message.Value);
    }
}
```

D.2.3 Publish/Subscribe

Contract:

```
public class Message : IEvent
{
    public string Msg { get; set; }
}
```

Server:

```
//endpoint configuration
public class EndpointConfig : IConfigureThisEndpoint, AsA_Publisher
{ }

//server that publishes something
public class Server : IWantToRunWhenBusStartsAndStops
{
    public IBus Bus { get; set; }

    public void Start()
    {
        //publish something...
        Bus.Publish(new Message { Msg = "test1" });
        ...
    }

    public void Stop()
    {
    }
}
```

Client:

```
//endpoint configuration
public class EndpointConfig : IConfigureThisEndpoint, AsA_Client
{ }

//handler for published messages
public class Client : IHandleMessages<Message>
{
    public void Handle(Message message)
    {
        Console.WriteLine("Notification received: " + message.Msg);
    }
}

//xml configuration for receiving notifications
<UnicastBusConfig>
    <MessageEndpointMappings>
```

```
      <add Messages="PubSub.Contract" Type="PubSub.Contract.Message"
          Endpoint="PubSub.Server" />
   </MessageEndpointMappings>
</UnicastBusConfig>
```

D.2.4 Usability Evaluation Tables

Table D.4: Evaluation of NServiceBus: Simple message transfer

Search	Usage	Sum	Concept	Usages	Description
30.0	29.4	59.4	ClassUsage	2	IConfigureThisEndpoint
30.0	21.0	51.0	ClassUsage	1	AsA_Client
30.0	21.0	51.0	ClassUsage	1	AsA_Server
30.0	21.0	51.0	ClassUsage	1	IWantToRunWhenBusStartsAndStops
30.0	21.0	51.0	ClassUsage	1	IBus
30.0	21.0	51.0	ClassUsage	1	IMessage
30.0	21.0	51.0	ClassUsage	1	IHandleMessages<>
0.0	16.8	16.8	InterfaceImpl	2	IConfigureThisEndpoint
0.0	12.0	12.0	InterfaceImpl	1	AsA_Client
0.0	12.0	12.0	InterfaceImpl	1	AsA_Server
0.0	12.0	12.0	InterfaceImpl	1	IWantToRunWhenBusStartsAndStops
0.0	12.0	12.0	InterfaceImpl	1	IHandleMessages<>
0.0	12.0	12.0	InterfaceImpl	1	IMessage
10.0	7.5	17.5	MemberImpl	1	IHandleMessages<>.Handle(Object)
16.5	20.5	37.0	MethodCall	1	IBus.Send(String,Object) (return value not used)
		0.0			InformationLookup Pattern: 0 / 8 InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.
		496.7	**Overall**		

Table D.5: Evaluation of NServiceBus: Request with multiple responses

Search	Usage	Sum	Concept	Usages	Description
30.0	29.4	59.4	ClassUsage	2	IConfigureThisEndpoint
30.0	21.0	51.0	ClassUsage	1	AsA_Client
30.0	21.0	51.0	ClassUsage	1	AsA_Server
30.0	21.0	51.0	ClassUsage	1	IWantToRunWhenBusStartsAndStops
30.0	29.4	59.4	ClassUsage	2	IBus
30.0	37.8	67.8	ClassUsage	3	IMessage
30.0	37.8	67.8	ClassUsage	3	IHandleMessages<>
0.0	16.8	16.8	InterfaceImpl	2	IConfigureThisEndpoint
0.0	12.0	12.0	InterfaceImpl	1	AsA_Client
0.0	12.0	12.0	InterfaceImpl	1	AsA_Server
0.0	12.0	12.0	InterfaceImpl	1	IWantToRunWhenBusStartsAndStops
0.0	12.0	12.0	InterfaceImpl	1	IHandleMessages<>
0.0	21.6	21.6	InterfaceImpl	3	IMessage
10.0	13.5	23.5	MemberImpl	3	IHandleMessages<>.Handle(Object)
16.5	20.5	37.0	MethodCall	1	IBus.Send(String,Object) (return value not used)
					InformationLookup Pattern: 0 / 8
11.5	10.5	22.0	MethodCall	2	IBus.Reply(Object)
		0.0			InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.
		576.3	**Overall**		

Table D.6: Evaluation of NServiceBus: Publish/subscribe

Search	Usage	Sum	Concept	Usages	Description
30.0	29.4	59.4	ClassUsage	2	IConfigureThisEndpoint
30.0	21.0	51.0	ClassUsage	1	AsA_Client
30.0	21.0	51.0	ClassUsage	1	AsA_Publisher
30.0	21.0	51.0	ClassUsage	1	IWantToRunWhenBusStartsAndStops
30.0	21.0	51.0	ClassUsage	1	IBus
30.0	21.0	51.0	ClassUsage	1	IEvent
30.0	21.0	51.0	ClassUsage	1	IHandleMessages<>
0.0	16.8	16.8	InterfaceImpl	2	IConfigureThisEndpoint
0.0	12.0	12.0	InterfaceImpl	1	AsA_Client
0.0	12.0	12.0	InterfaceImpl	1	AsA_Publisher
0.0	12.0	12.0	InterfaceImpl	1	IWantToRunWhenBusStartsAndStops
0.0	12.0	12.0	InterfaceImpl	1	IHandleMessages<>
0.0	12.0	12.0	InterfaceImpl	1	IEvent
10.0	7.5	17.5	MemberImpl	1	IHandleMessages<>.Handle(Object)
15.5	22.5	38.0	MethodCall	3	IBus.Publish<>(Object)
		15.0			Xml Pattern XML doesn't have very good usability and should only be used when it is necessary to change the configuration without recompiling. Otherwise consider e.g. using annotations or a fluent interface instead. Consider making a schema available to enable auto completion in the XML.
12.0	19.5	31.5	Xml	1	XML element: <UnicastBusConfig /> The name UnicastBusConfig is rather long, especially when there is no auto completion available - consider making it shorter.
12.0	19.5	31.5	Xml	1	XML element: <MessageEndpointMappings /> The name MessageEndpointMappings is rather long, especially when there is no auto completion available - consider making it shorter.
39.0	70.5	109.5	Xml	1	XML element: <add Messages Type Endpoint />
		685.2	**Overall**		

D.3 Application Space

D.3.1 Simple Message Transfer

Contract:
No contract is required for this scenario.

Server:

```
using (var space = new XcoAppSpace("tcp.port=8000"))
{
   space.Run<string>(m => Console.WriteLine("Msg received: " + m));
   Console.ReadLine();
}
```

Client:

```
using (var space = new XcoAppSpace("tcp.port=0"))
{
   var port = space.Connect<string>("localhost:8000");
   port.Post("Hello, World!");

   Console.ReadLine();
}
```

D.3.2 Request with Multiple Responses

Contract:

```
[Serializable]
public class Message
{
   public string Msg { get; set; }
   public Port<int> ProgressPort { get; set; }
   public Port<string> ResponsePort { get; set; }
}
```

Server:

```
using (var space = new XcoAppSpace("tcp.port=8000"))
{
   space.Run<Message>(m =>
   {
      var result = WorkProcessor.Process(m.Msg, progress =>
          m.ProgressPort.Post(progress));
      m.ResponsePort.Post(result);
   });
   Console.ReadLine();
}
```

Client:

```
using (var space = new XcoAppSpace("tcp.port=0"))
{
   var port = space.Connect<Message>("localhost:8000");
   var progressPort = space.Receive<int>(m =>
      Console.WriteLine("Progress: " + m));
   var responsePort = space.Receive<string>(m =>
      Console.WriteLine("Response: " + m));
   port.Post(new Message{Msg = "testmsg", ProgressPort =
      progressPort, ResponsePort = responsePort});

   Console.WriteLine("Message sent.");
   Console.ReadLine();
}
```

D.3.3 Publish/Subscribe

Contract:

```
public class NotificationWorker : PortSet<string, Subscribe<string>,
   Unsubscribe<string>>
{ }
```

Server:

```
//worker implementation
public class NotificationWorkerImpl : NotificationWorker
{
   private readonly XcoPublisher<string> publisher =
      new XcoPublisher<string>();

   [XcoConcurrent]
   void Process(string msg)
   {
      publisher.Publish(WorkProcessor.Process(msg));
   }
}

//main method
using (var space = new XcoAppSpace("tcp.port=8000"))
{
   var worker = space.RunWorker<NotificationWorker,
      NotificationWorkerImpl>();

   //public something...
   worker.Post("notification1");
   ...

   Console.ReadLine();
}
```

Client:

```
using (var space = new XcoAppSpace("tcp.port=0"))
{
   var worker =
      space.ConnectWorker<NotificationWorker>("localhost:8000");
   var subscriptionPort = space.Receive<string>(m =>
      Console.WriteLine("Notification: " + m));
```

```
    worker.Post(new Subscribe<string>(subscriptionPort));
    Console.WriteLine("subscribed");
    Console.ReadLine();

    worker.Post(new Unsubscribe<string>(subscriptionPort));
    Console.WriteLine("unsubscribed");
    Console.ReadLine();
}
```

D.3.4 Usability Evaluation Tables

Table D.7: Evaluation of AppSpace: Simple message transfer

Search	Usage	Sum	Concept	Usages	Description
30.0	29.4	59.4	ClassUsage	2	XcoAppSpace
2.0	9.1	11.1	Instantiation	2	new XcoAppSpace(String)
15.0	7.5	22.5	MethodCall	1	XcoAppSpace.Run<>(Handler<>) (return value not used)
14.0	23.5	37.5	MethodCall	1	XcoAppSpace.Connect<>(String) InformationLookup Pattern: 0 / 8
10.7	7.5	18.2	MethodCall	1	Port<>.Post(String)
9.0	11.3	20.3	ConfigString	2	Config string: tcp.port=8000 (2 usages)
		0.0			InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.
		169.0	**Overall**		

Table D.8: Evaluation of AppSpace: Request with multiple responses

Search	Usage	Sum	Concept	Usages	Description
30.0	29.4	59.4	ClassUsage	2	XcoAppSpace
30.0	29.4	59.4	ClassUsage	2	Port<>
30.0	29.0	59.0	ClassUsage	1	SerializableAttribute
2.0	9.1	11.1	Instantiation	2	new XcoAppSpace(String)
0.0	4.0	4.0	Instantiation	1	[SerializableAttribute()]
15.0	7.5	22.5	MethodCall	1	XcoAppSpace.Run<>(Handler<>) (return value not used)
14.0	23.5	37.5	MethodCall	1	XcoAppSpace.Connect<>(String) InformationLookup Pattern: 0 / 8
19.0	21.7	40.7	MethodCall	2	XcoAppSpace.Receive<>(Handler<>)
10.7	13.5	24.2	MethodCall	3	Port<>.Post(String)
		0.0			ConfigString Pattern
9.0	11.3	20.3	ConfigString	2	Config string: tcp.port=8000 (2 usages)
		0.0			InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.
		338.1	**Overall**		

Table D.9: Evaluation of AppSpace: Publish/subscribe

Search	Usage	Sum	Concept	Usages	Description
30.0	29.4	59.4	ClassUsage	2	XcoAppSpace
30.0	21.0	51.0	ClassUsage	1	PortSet<,,>
30.0	21.0	51.0	ClassUsage	1	XcoPublisher<>
30.0	29.4	59.4	ClassUsage	2	Subscribe<>
30.0	29.4	59.4	ClassUsage	2	Unsubscribe<>
30.0	29.0	59.0	ClassUsage	1	XcoConcurrentAttribute
2.0	9.1	11.1	Instantiation	2	new XcoAppSpace(String)
21.5	12.5	34.0	MethodCall	1	XcoAppSpace.RunWorker<,>() (return value not used)
17.5	28.5	46.0	MethodCall	1	XcoAppSpace.ConnectWorker<>(String) InformationLookup Pattern: 0 / 8
19.0	15.5	34.5	MethodCall	1	XcoAppSpace.Receive<>(Handler<>)
10.7	19.5	30.2	MethodCall	5	Port<>.Post(String)
0.0	12.0	12.0	Inheritance	1	new PortSet<,,>()
0.0	4.0	4.0	Instantiation	1	new XcoPublisher<>()
10.2	7.5	17.7	MethodCall	1	XcoPublisher<>.Publish(String)
2.0	6.5	8.5	Instantiation	1	new Subscribe<>(Port<>)
2.0	6.5	8.5	Instantiation	1	new Unsubscribe<>(Port<>)
9.0	11.3	20.3	ConfigString	2	Config string: tcp.port=8000 (2 usages)
		0.0			InformationLookup Pattern: Some concepts require lookup of external information - this should be prevented where possible.
		565.9	**Overall**		

Bibliography

[1] Allan J. Albrecht. Measuring application development productivity. *IBM Applications Development Symposium*, pages 83–92, 1979.

[2] Allan J. Albrecht and John E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *Software Engineering, IEEE Transactions on*, SE-9(6):639–648, 1983.

[3] Alexandre Alvaro, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A software component quality framework. *SIGSOFT Softw. Eng. Notes*, 35:1–18, January 2010.

[4] ANSI. *Common Industry Format for Usability Test Reports (ANSI-NCITS 354-2001)*. ANSI, 2001.

[5] Ken Arnold. Programmers are people, too. *Queue*, 3(5):54–59, June 2005.

[6] B. Auprasert and Y. Limpiyakorn. Structuring cognitive information for software complexity measurement. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 7, pages 830–834, 2009.

[7] RobertW. Bailey, CariA. Wolfson, Janice Nall, and Sanjay Koyani. Performance-based usability testing: Metrics that have the greatest impact for improving a system's usability. In Masaaki Kurosu, editor, *Human Centered Design*, volume 5619 of *Lecture Notes in Computer Science*, pages 3–12. Springer Berlin Heidelberg, 2009.

[8] R.K. Bandi, V.K. Vaishnavi, and D.E. Turk. Predicting maintenance performance using object-oriented design complexity metrics. *Software Engineering, IEEE Transactions on*, 29(1):77–87, 2003.

[9] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28:4–17, January 2002.

[10] Barisits, Martin-Stefan. Design and Implementation of the next Generation XVSM Framework - Operations, Coordination and Transaction. Master's thesis, Vienna University of Technology, 2010.

[11] Steffen Bartsch. Authorization enforcement usability case study. In Ulfar Erlingsson, Roel Wieringa, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, volume 6542 of *Lecture Notes in Computer Science*, pages 209–220. Springer Berlin Heidelberg, 2011.

[12] Jack K. Beaton, Brad A. Myers, Jeffrey Stylos, Sae Young (Sophie) Jeong, and Yingyu (Clare) Xie. Usability evaluation for enterprise soa apis. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, SDSOA '08, pages 29–34, New York, NY, USA, 2008. ACM.

[13] Manuel F. Bertoa, José M. Troya, and Antonio Vallecillo. Measuring the usability of software components. *J. Syst. Softw.*, 79:427–439, March 2006.

[14] Manuel F. Bertoa and Antonio Vallecillo. Quality attributes for cots components. *I+ D Computacion, 1 (2)*, pages 128–143, 2002.

[15] Nigel Bevan. Extending quality in use to provide a framework for usability measurement. In Masaaki Kurosu, editor, *Human Centered Design*, volume 5619 of *Lecture Notes in Computer Science*, pages 13–22. Springer Berlin Heidelberg, 2009.

[16] Nigel Bevan and Miles MacLeod. Usability measurement in context. *Behaviour & Information Technology*, 13(1-2):132–145, 1994.

[17] Randolph G. Bias. The pluralistic usability walkthrough: coordinated empathies. In Jakob Nielsen and Robert L. Mack, editors, *Usability inspection methods*, pages 63–76. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[18] Alan Blackwell and Thomas Green. A cognitive dimensions questionnaire optimised for users. In *Proceedings of the Twelfth Annual Meeting of the Psychology of Programming Interest Group*, pages 137–152, 2000.

[19] Alan Blackwell and Thomas Green. Notational systems–the cognitive dimensions of notations framework. *HCI Models, Theories, and Frameworks: Toward an Interdisciplinary Science*, 2003.

[20] Joshua Bloch. How to design a good api and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 506–507, New York, NY, USA, 2006. ACM.

[21] OpenHallway Blog. What is hallway usability testing? `http://blog.openhallway.com/?p=146`. Accessed: 2013-06-19.

[22] L.C. Briand, Sandro Morasca, and V.R. Basili. Property-based software engineering measurement. *Software Engineering, IEEE Transactions on*, 22(1):68–86, 1996.

[23] W.-P. Brinkman, R. Haakma, and D. G. Bouwhuis. The theoretical foundation and validity of a component-based usability questionnaire. *Behaviour & Information Technology*, 28(2):121–137, 2009.

[24] F. Brito e Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, pages 90–99, 1996.

[25] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Softw.*, 15:37–46, September 1998.

[26] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[27] Andreas Brückl. Relaxed non-blocking distributed transactions for the eXtensible virtual shared memory. Master's thesis, Vienna University of Technology, 2013.

[28] J.C. Cherniavsky and C.H. Smith. On weyuker's axioms for software complexity measures. *Software Engineering, IEEE Transactions on*, 17(6):636–638, 1991.

[29] Jitender Kumar Chhabra, K.K. Aggarwal, and Yogesh Singh. Code and data spatial complexity: two important software understandability measures. *Information and Software Technology*, 45(8):539 – 546, 2003.

[30] Jitender Kumar Chhabra and Varun Gupta. Evaluation of object-oriented spatial complexity measures. *SIGSOFT Softw. Eng. Notes*, 34(3):1–5, May 2009.

[31] Yam Bahadur Chhetri. *Classifying and Recommending Knowledge in Reference Documentation to Improve API Usability*. PhD thesis, McGill University, 2012.

[32] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493, June 1994.

[33] John P. Chin, Virginia A. Diehl, and Kent L. Norman. Development of an instrument measuring user satisfaction of the human-computer interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '88, pages 213–218, New York, NY, USA, 1988. ACM.

[34] Eun Sook Cho, Min Sun Kim, and Soo Dong Kim. Component metrics to measure component quality. In *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, APSEC '01, pages 419–, Washington, DC, USA, 2001. IEEE Computer Society.

[35] Steven Clarke. Attributes and api usability (again!). `http://blogs.msdn.com/b/stevencl/archive/2004/10/08/239833.aspx`, October 2004.

[36] Steven Clarke. Attributes and api usability revisited. `http://blogs.msdn.com/b/stevencl/archive/2004/05/12/130826.aspx`, May 2004.

[37] Steven Clarke. Measuring API usability. *Dr. Dobb's Journal*, 29:S6–S9, 2004.

[38] Steven Clarke. How usable are your apis? In Andy Oram and Greg Wilson, editors, *Making Software: What Really Works, and Why We Believe It*. O'Reilly Media, 2010.

[39] Steven Clarke and Curtis Becker. Using the Cognitive Dimensions Framework to evaluate the usability of a class library. In M. Petre and D. Budgen, editors, *Proc. of Joint Conf. EASE & PPIG 2003*, 2003.

[40] Larry L. Constantine and Lucy A.D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Pearson Education, 1999.

[41] Corillian Corporation. NDepend Placemat, 2007.

[42] Stefan Craß. A formal model of the extensible virtual shared memory (xvsm) and its implementation in haskell – design and specification. Master's thesis, Vienna University of Technology, 2010.

[43] Stefan Craß, eva Kühn, and Gernot Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *Int. Database Engineering and Applications Symposium (IDEAS)*, ACM, pages 301–306, 2009.

[44] Douglas Crockford. The application/json media type for javascript object notation (json), 2006.

[45] Krzysztof Cwalina and Brad Abrams. *Framework design guidelines: conventions, idioms, and patterns for reusable .net libraries*. Addison-Wesley Professional, first edition, 2005.

[46] C.R.B. de Souza and D.L.M. Bentolila. Automatic evaluation of api usability using complexity metrics and visualizations. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 299–302, 2009.

[47] Uri Dekel and J.D. Herbsleb. Improving api documentation usability with knowledge pushing. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 320–330, 2009.

[48] Dönz, Tobias. Design and Implementation of the next Generation XVSM Framework - Runtime, Protocol and API. Master's thesis, Vienna University of Technology, 2011.

[49] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1st edition, 1998.

[50] Ekwa Duala-Ekoko and Martin P. Robillard. The information gathering strategies of api learners. Technical report, School of Computer Science, McGill University, 2010.

[51] Ekwa Duala-Ekoko and Martin P. Robillard. Using structure-based recommendations to facilitate discoverability in apis. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 79–104, Berlin, Heidelberg, 2011. Springer-Verlag.

[52] Ekwa Duala-Ekoko and Martin P. Robillard. Asking and answering questions about unfamiliar apis: an exploratory study. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 266–276, Piscataway, NJ, USA, 2012. IEEE Press.

[53] Sanjay Kumar Dubey and Ajay Rana. Assessment of usability metrics for object-oriented software system. *SIGSOFT Softw. Eng. Notes*, 35:1–4, December 2010.

[54] Sanjay Kumar Dubey and Ajay Rana. Usability estimation of software system by using object-oriented metrics. *SIGSOFT Softw. Eng. Notes*, 36(2):1–6, May 2011.

[55] Joseph S Dumas and Janice C Redish. *A practical guide to usability testing*. Intellect Ltd, 1999.

[56] Daniel S. Eisenberg, Jeffrey Stylos, A. Faulring, and Brad A. Myers. Using association metrics to help users navigate api documentation. In *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*, pages 23–30, 2010.

[57] Daniel S. Eisenberg, Jeffrey Stylos, and Brad A. Myers. Apatite: a new interface for exploring apis. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1331–1334, New York, NY, USA, 2010. ACM.

[58] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proc. of the 29th Int. Conf. on Softw. Eng.*, ICSE '07, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society.

[59] eva Kühn and Vesna Sesum-Cavic. A space-based generic pattern for self-initiative load balancing agents. In Huib Aldewereld, Virginia Dignum, and Gauthier Picard, editors, *ESAW*, volume 5881 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2009.

[60] Umer Farooq, Leon Welicki, and Dieter Zirkler. API usability peer reviews: a method for evaluating the usability of application programming interfaces. In *Proc. of the 28th Int. Conf. on Human Factors in Computing Systems*, CHI '10, pages 2327–2336, New York, NY, USA, 2010. ACM.

[61] L. Feldman, C.J. Mueller, D. Tamir, and O.V. Komogortsev. Usability testing with total-effort metrics. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 426–429, 2009.

[62] N.E. Fenton and M. Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675–689, 1999.

[63] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[64] Andy Field, Jeremy Miles, and Zoe Field. *Discovering Statistics Using R*. SAGE Publications, 2012.

[65] Sally Fincher. Patterns for hci and cognitive dimensions: two halves of the same story? In J. Kuljis, L. Baldwin, and R. Scoble, editors, *Proceedings of the Fourteenth Annual Workshop of the Psychology of Programming Interest Group*, pages 156–172, 2002.

[66] R.A. Fisher. *Statistical methods for research workers*. Biological monographs and manuals. Oliver and Boyd, 1932.

[67] Eelke Folmer and Jan Bosch. Architecting for usability: a survey. *Journal of Systems and Software*, 70(1-2):61 – 78, 2004.

[68] The R Project for Statistical Computing. `http://www.r-project.org`. Accessed: 2013-05-31.

[69] Martin Fowler. Cannot measure productivity. `http://martinfowler.com/bliki/CannotMeasureProductivity.html`, August 2003.

[70] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. `http://martinfowler.com/articles/injection.html`, January 2004.

[71] Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley Signature Series. Addison-Wesley, 2010.

[72] John E. Gaffney. Estimating the number of faults in code. *Software Engineering, IEEE Transactions on*, SE-10(4):459–464, 1984.

[73] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[74] David Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[75] Matt Gemmell. Api design. `http://mattgemmell.com/2012/05/24/api-design`. Accessed: 2013-07-26.

[76] Leopoldo A. Gemoets and Mo Adam Mahmood. Effect of the quality of user documentation on user satisfaction with information systems. *Inf. Manage.*, 18(1):47–54, January 1990.

[77] Jill Gerhardt-Powals. Cognitive engineering principles for enhancing human-computer performance. *International Journal of Human-Computer Interaction*, 8(2):189–211, 1996.

[78] Jens Gerken, Hans-Christian Jetter, Michael Zöllner, Martin Mader, and Harald Reiterer. The concept maps method as a tool to evaluate the usability of apis. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 3373–3382, New York, NY, USA, 2011. ACM.

[79] Nasib S. Gill and Balkishan. Dependency and interaction oriented complexity metrics of component-based systems. *SIGSOFT Softw. Eng. Notes*, 33:3:1–3:5, March 2008.

[80] Nasib S. Gill and P. S. Grover. Few important considerations for deriving interface complexity metric for component-based systems. *SIGSOFT Softw. Eng. Notes*, 29:4–4, March 2004.

[81] Joris Graaumans. A qualitative study to the usability of three xml query languages. In *Proc. of the conference on Dutch directions in HCI*, Dutch HCI '04, pages 6–9, New York, NY, USA, 2004. ACM.

[82] Wayne D. Gray and Marilyn C. Salzman. Damaged merchandise? a review of experiments that compare usability evaluation methods. *Human-Computer Interaction*, 13(3):203–261, 1998.

[83] T.R.G. Green, A.E. Blandford, L. Church, C.R. Roast, and S. Clarke. Cognitive dimensions: Achievements, new directions, and open questions. *Journal of Visual Languages & Computing*, 17(4):328–365, 2006.

[84] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131 – 174, 1996.

[85] Thomas Grill, Ondrej Polacek, and Manfred Tscheligi. Methods towards api usability: A structural analysis of usability problem categories. In Marco Winckler, Peter Forbrig, and Regina Bernhaupt, editors, *Human-Centered Software Engineering*, volume 7623 of *Lecture Notes in Computer Science*, pages 164–180. Springer Berlin Heidelberg, 2012.

[86] Nielsen Norman Group. How Many Test Users in a Usability Study? `http://www.nngroup.com/articles/how-many-test-users`. Accessed: 2013-05-29.

[87] Nielsen Norman Group. Putting A/B Testing in Its Place. `http://www.nngroup.com/articles/putting-ab-testing-in-its-place`. Accessed: 2013-05-29.

[88] The Guardian. Microsoft cuts 'Mr Clippy'. `http://www.guardian.co.uk/media/2001/apr/11/advertising2`. Accessed: 2013-06-05.

[89] Maurice H. Halstead. *Elements of software science (Operating and programming systems series)*. Elsevier, 1977.

[90] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion from abbreviated input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 332–343, Washington, DC, USA, 2009. IEEE Computer Society.

[91] H. Rex Hartson, José C. Castillo, John Kelso, and Wayne C. Neale. Remote evaluation: the network as an extension of the usability laboratory. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '96, pages 228–235, New York, NY, USA, 1996. ACM.

[92] Reinhold Hatzinger, Kurt Hornik, and Herbert Nagel. *R: Einführung durch angewandte Statistik*. Pearson Studium - Scientific Tools. Pearson Studium, 2011.

[93] Lars Heinemann and Benjamin Hummel. Recommending api methods based on identifier contexts. In *Proceeding of the 3rd international workshop on Search-driven development: users, infrastructure, tools, and evaluation*, SUITE '11, pages 1–4, New York, NY, USA, 2011. ACM.

[94] Michi Henning. API design matters. *Queue*, 5:24–36, May 2007.

[95] Sallie Henry and Dennis Kafura. The evaluation of software systems' structure using quantitative software metrics. *Software: Practice and Experience*, 14(6):561–573, 1984.

[96] Morten Hertzum and Niels E. Jacobsen. The Evaluator Effect: A Chilling Fact About Usability Evaluation Methods. *Int. Journal of Human-Computer Interaction*, 15(1):183–204, 2003.

[97] Jürgen Hirsch. An adaptive and flexible replication mechanism for mozartspaces, the xvsm reference implementation. Master's thesis, Vienna University of Technology, 2012.

[98] Tasha Hollingsed and David G. Novick. Usability inspection methods after 15 years of research and practice. In *Proceedings of the 25th annual ACM international conference on Design of communication*, SIGDOC '07, pages 249–255, New York, NY, USA, 2007. ACM.

[99] Daqing Hou and David M. Pletcher. Towards a better code completion system by api grouping, filtering, and popularity-based ranking. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 26–30, New York, NY, USA, 2010. ACM.

[100] Ebba Thora Hvannberg, Effie Lai-Chong Law, and Marta Kristín Lárusdóttir. Heuristic evaluation: Comparing ways of finding and reporting usability problems. *Interact. Comput.*, 19:225–240, March 2007.

[101] IEEE. *IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[102] ISO/IEC. *ISO/IEC 9241-11: Guidance on Usability*. ISO/IEC, 1998.

[103] ISO/IEC. *ISO/IEC 9126: Software engineering – Product quality*. ISO/IEC, 2001.

[104] ISO/IEC. *ISO/IEC 25010: Systems and software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Quality models for software product quality and system quality in use*. ISO/IEC, 2011.

[105] Robin Jeffries, James R. Miller, Cathleen Wharton, and Kathy Uyeda. User interface evaluation in the real world: a comparison of four techniques. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 119–124, New York, NY, USA, 1991. ACM.

[106] Robert J Johnston, Thomas F Weaver, Lynn A Smith, Stephen K Swallow, et al. Contingent valuation focus groups: insights from ethnographic interview techniques. *Agricultural and Resource Economics Review*, 24(1):56–69, 1995.

[107] Sivamuni Kalaimagal and Rengaramanujam Srinivasan. A retrospective on software component quality models. *SIGSOFT Softw. Eng. Notes*, 33(6):1–10, October 2008.

[108] L. Kantner, R. Shroyer, and S. Rosenbaum. Structured heuristic evaluation of online documentation. In *Professional Communication Conference, 2002. IPCC 2002. Proceedings. IEEE International*, pages 331–342, 2002.

[109] Laurie Kantner, Deborah Hinderer Sova, and Stephanie Rosenbaum. Alternative methods for field usability research. In *Proceedings of the 21st annual international conference on Documentation*, SIGDOC '03, pages 68–72, New York, NY, USA, 2003. ACM.

[110] Claire-Marie Karat, Robert Campbell, and Tarra Fiegel. Comparison of empirical testing and walkthrough methods in user interface evaluation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 397–404, New York, NY, USA, 1992. ACM.

[111] Latika Kharb and Rajender Singh. Complexity metrics for component-oriented software systems. *SIGSOFT Softw. Eng. Notes*, 33:4:1–4:3, March 2008.

[112] Jurek Kirakowski and Mary Corbett. Sumi: the software usability measurement inventory. *British Journal of Educational Technology*, 24(3):210–212, 1993.

[113] B.A. Kitchenham, L.M. Pickard, and S.J. Linkman. An evaluation of some design metrics. *Software Engineering Journal*, 5(1):50–58, 1990.

[114] A.J. Ko and Y. Riche. The role of conceptual knowledge in api usability. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 173–176, 2011.

[115] Eva Kühn, Stefan Crass, Gerson Joskowicz, Alexander Marek, and Thomas Scheller. Peer-based programming model for coordination patterns. In Rocco Nicola and Christine Julien, editors, *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin Heidelberg, 2013.

[116] eva Kühn, Alexander Marek, Thomas Scheller, Vesna Sesum-Cavic, Michael Vögler, and Stefan Craß. A space-based generic pattern for self-initiative load clustering agents. In *14th Int. Conf. on Coordination Models and Languages (COORDINATION)*, volume 7274 of *LNCS*, pages 230–244. Springer, 2012.

[117] eva Kühn, Richard Mordinyi, Laszlo Keszthelyi, and Christian Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automation domain. In *8th Int'l Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 625–632. IFAAMAS, 2009.

[118] Dharmender Singh Kushwaha and A. K. Misra. Improved cognitive information complexity measure: a metric that establishes program comprehension effort. *SIGSOFT Softw. Eng. Notes*, 31(5):1–7, September 2006.

[119] Dharmender Singh Kushwaha and A.K. Misra. Evaluating cognitive information complexity measure. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 2 pp.–504, 2006.

[120] V. Lakshmi Narasimhan and B. Hendradjaya. Some theoretical considerations for a suite of metrics for the integration of software components. *Inf. Sci.*, 177:844–864, February 2007.

[121] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, ast-based api-usage analysis of open-source java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1317–1324, New York, NY, USA, 2011. ACM.

[122] Bernhard Lauer. dotnetpro-rankings. *dotNetPro*, 2013(7):139–140, July 2013.

[123] Bettina Laugwitz, Theo Held, and Martin Schrepp. Construction and evaluation of a user experience questionnaire. In Andreas Holzinger, editor, *HCI and Usability for Education and Work*, volume 5298 of *Lecture Notes in Computer Science*, pages 63–76. Springer Berlin Heidelberg, 2008.

[124] Clayton Lewis and Cathleen Wharton. Cognitive walkthroughs. *Handbook of human-computer interaction*, 2:717–732, 1997.

[125] James R. Lewis. Ibm computer usability satisfaction questionnaires: Psychometric evaluation and instructions for use. *International Journal of Human-Computer Interaction*, 7(1):57–78, 1995.

[126] Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*. Psychology of Programming Interest Group, 2006.

[127] M. Lipow. Number of faults per line of code. *Software Engineering, IEEE Transactions on*, SE-8(4):437–439, 1982.

[128] Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[129] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM.

[130] Robert C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[131] Neil McAllister. The futility of developer productivity metrics. http://www.infoworld.com/d/application-development/the-futility-developer-productivity-metrics-179244, November 2011.

[132] Thomas J. McCabe. A complexity measure. In *Proc. of the 2nd Int. Conf. on Software engineering*, ICSE '76, pages 308–320, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[133] Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. Building more usable apis. *IEEE Softw.*, 15:78–86, May 1998.

[134] J. Michura and L.F. Capretz. Metrics suite for class complexity. In *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, volume 2, pages 404–409 Vol. 2, 2005.

[135] S. Misra. Cognitive program complexity measure. In *Cognitive Informatics, 6th IEEE International Conference on*, pages 120–125, 2007.

[136] S. Misra. An object oriented complexity metric based on cognitive weights. In *Cognitive Informatics, 6th IEEE International Conference on*, pages 134–139, 2007.

[137] Sanjay Misra and Ibrahim Akman. Applicability of weyuker's properties on oo metrics: Some misunderstandings. *Computer Science and Information Systems/ComSIS*, 5(1):17–24, 2008.

[138] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

[139] M. Mooty, A. Faulring, J. Stylos, and B.A. Myers. Calcite: Completing code completion for constructors using crowds. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 15 –22, sept. 2010.

[140] Richard Mordinyi. *Managing Complex and Dynamic Software Systems with Space-Based Computing*. PhD thesis, Vienna University of Technology, 2010.

[141] Jakob Nielsen. Finding usability problems through heuristic evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 373–380. ACM, 1992.

[142] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, San Francisco, 1994.

[143] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. What programmers really want: results of a needs assessment for sdk documentation. In *Proceedings of the 20th annual international conference on Computer documentation*, SIGDOC '02, pages 133–141, New York, NY, USA, 2002. ACM.

[144] Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 54:1–54:11, New York, NY, USA, 2012. ACM.

[145] Hector M. Olague, Letha H. Etzkorn, Sherri L. Messimer, and Harry S. Delugach. An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(3):171–197, 2008.

[146] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344, 1992.

[147] C. Omar, YoungSeok Yoon, T.D. LaToza, and B.A. Myers. Active code completion. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 261 –262, sept. 2011.

[148] Steven Pemberton. Programmers are humans too. *SIGCHI Bulletin*, 29, July 1997.

[149] Kara Pernice and Jakob Nielsen. *Eyetracking Methodology*. Nielsen Norman Group, 2009.

[150] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. An empirical study of api usability. Technical report, ETH Zurich, 2013.

[151] David M. Pletcher and Daqing Hou. BCC: Enhancing code completion for better API usability. In *Proceeings of the 25th IEEE International Conference on Software Maintenance*, ICSM '09, pages 393–394, Edmonton, Alberta, Canada, 2009. IEEE.

[152] Peter G. Polson, Clayton Lewis, John Rieman, and Cathleen Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. *Int. J. Man-Mach. Stud.*, 36:741–773, May 1992.

[153] Girish Maskeri Rama and Avinash Kak. Some structural measures of api usability. *Software: Practice and Experience*, pages n/a–n/a, 2013.

[154] Dominik Rauch. Peerspace.net – implementing and evaluating the peer model with focus on api usability. Master's thesis, Vienna University of Technology, 2014.

[155] Jeffrey Richter. Concurrent Affairs: Concurrency and Coordination Runtime. *MSDN Magazine*, September 2006.

[156] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 317–326, Washington, DC, USA, 2008. IEEE Computer Society.

[157] Martin P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Softw.*, 26:27–34, November 2009.

[158] MartinP. Robillard and Robert DeLine. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[159] Gernot Rumpold. Analysis on API-usability for space-based computing frameworks. Project report at Institute of Computer Languages, Vienna University of Technology, 2014.

[160] C.R. Rupakheti and Daqing Hou. Satisfying programmers' information needs in api-based programming. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 250–253, 2011.

[161] Filipp Sapienza. Usability, structured content, and single sourcing with xml. *Technical Communication*, 51(3):399–408, 2004.

[162] Jeff Sauro and James R. Lewis. Correlations among prototypical usability metrics: evidence for the construct of usability. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1609–1618, New York, NY, USA, 2009. ACM.

[163] Christopher Scaffidi. Why are apis difficult to learn and use? *Crossroads*, 12(4):4–4, August 2006.

[164] Thomas Scheller. Einfach und flexibel: Application Space. *Visual Studio One*, 5:20–24, 2010.

[165] Thomas Scheller. Open Source: Application Space. *Visual Studio One*, 6:22–25, 2010.

[166] Thomas Scheller. Application Space: Verteilte Software für Normalsterbliche. *Visual Studio One*, 1:30–34, 2012.

[167] Thomas Scheller and eva Kühn. Measurable concepts for the usability of software components. In *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA '11, pages 129–133, Oulu, Finland, 2011. IEEE Computer Society.

[168] Thomas Scheller and eva Kühn. Influencing factors on the usability of api classes and methods. In *19th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems*, ECBS '12, pages 232–241, Novi Sad, Serbia, 2012. IEEE Computer Society.

[169] Thomas Scheller and eva Kühn. Influence of code completion methods on API usability: A case study. In *The 12th IASTED International Conference on Software Engineering*, SE '13, pages 760–767, Innsbruck, Austria, 2013. ActaPress.

[170] Thomas Scheller and Eva Kühn. Usability evaluation of configuration-based api design concepts. In Andreas Holzinger, Martina Ziefle, Martin Hitz, and Matjaz Debevc, editors, *Human Factors in Computing and Informatics*, volume 7946 of *Lecture Notes in Computer Science*, pages 54–73. Springer Berlin Heidelberg, 2013.

[171] Thomas Scheller and eva Kühn. Automated measurement of api usability: The api concepts framework. In *submitted for journal publication*, 2014.

[172] S. Sedigh-Ali, A. Ghafoor, and R.A. Paul. Software engineering metrics for cots-based systems. *Computer*, 34(5):44–50, 2001.

[173] Robert L. Sedlmeyer, Joseph K. Kearney, William B. Thompson, Michael A. Adler, and Michael A. Gray. Problems with software complexity measurement. In *Proceedings of the 1985 ACM thirteenth annual conference on Computer Science*, CSC '85, pages 340–347, New York, NY, USA, 1985. ACM.

[174] Reinhard Sefelin, Manfred Tscheligi, and Verena Giller. Paper prototyping - what is it good for?: a comparison of paper- and computer-based low-fidelity prototyping. In *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '03, pages 778–779, New York, NY, USA, 2003. ACM.

[175] Ahmed Seffah, Mohammad Donyaee, Rex B. Kline, and Harkirat K. Padda. Usability measurement and metrics: A consolidated model. *Software Quality Control*, 14:159–178, 2006.

[176] Arun Sharma, Rajesh Kumar, and P. S. Grover. Empirical evaluation and critical review of complexity metrics for software components. In *Proc. of the 6th WSEAS Int. Conf. on Softw. Eng., Parallel and Distributed Systems*, pages 24–29, Stevens Point, Wisconsin, USA, 2007. WSEAS.

[177] Naveen Sharma, Padmaja Joshi, and P. Joshi. Applicability of weyuker's property 9 to object oriented metrics. *Software Engineering, IEEE Transactions on*, 32(3):209–211, 2006.

[178] Regis P.S. Simao and Arnaldo D. Belchior. Quality characteristics for software components: Hierarchy and quality guides. In Alejandra Cechich, Mario Piattini, and Antonio Vallecillo, editors, *Component-Based Software Quality*, volume 2693 of *Lecture Notes in Computer Science*, pages 184–206. Springer Berlin Heidelberg, 2003.

[179] Patrick Smacchia. NDepend. `http://www.ndepend.com`. Accessed: 2013-05-29.

[180] Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 269–287, Berlin, Heidelberg, 2012. Springer-Verlag.

[181] J. Stylos, A. Faulring, Zizhuang Yang, and B.A. Myers. Improving api documentation using api usage information. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, pages 119–126, 2009.

[182] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proc. of the 29th Int. Conf. on Softw. Eng.*, ICSE '07, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society.

[183] Jeffrey Stylos and Brad Myers. Mapping the space of api design decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC 2007, pages 50–60. IEEE Computer Society, 2007.

[184] Jeffrey Stylos and Brad Myers. The implications of method placement on API learnability. In *Proc. of the 16th ACM SIGSOFT Int. Symposium on Foundations of Softw. Eng.*, SIGSOFT '08/FSE-16, pages 105–112, New York, NY, USA, 2008. ACM.

[185] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

[186] Dan E. Tamir, Oleg V. Komogortsev, Carl J. Mueller, Divya K.D. Venkata, Gregory R. LaKomski, and Arwa M. Jamnagarwala. Detection of software usability deficiencies. In Aaron Marcus, editor, *Design, User Experience, and Usability. Theory, Methods, Tools and Practice*, volume 6770 of *Lecture Notes in Computer Science*, pages 527–536. Springer Berlin Heidelberg, 2011.

[187] Dan E. Tamir and Carl J. Mueller. Pinpointing usability issues using an effort based framework. In *Systems Man and Cybernetics (SMC), 2010 IEEE International Conference on*, pages 931–938, 2010.

[188] Jaroslav Tulach. *Practical API Design: Confessions of a Java Framework Architect*. Apress, Berkely, CA, USA, first edition, 2008.

[189] P.R. Vora and M.G. Helander. A teaching method as an alternative to the concurrent think-aloud method for usability testing. In Katsuhiko Ogawa Yuichiro Anzai and Hirohiko Mori, editors, *Symbiosis of Human and Artifact Future Computing and Design for Human-Computer Interaction Proceedings of the Sixth International Conference on Human-Computer Interaction, (HCI International '95)*, volume 20 of *Advances in Human Factors/Ergonomics*, pages 375 – 380. Elsevier, 1995.

[190] Yingxu Wang and Jingqiu Shao. Measurement of the cognitive functional complexity of software. In *Cognitive Informatics, 2003. Proceedings. The Second IEEE International Conference on*, pages 67–74, 2003.

[191] Hironori Washizaki, T. Nakagawa, Y. Saito, and Y. Fukazawa. A coupling-based complexity metric for remote component-based software systems toward maintainability estimation. In *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, pages 79–86, 2006.

[192] Hironori Washizaki, Hirokazu Yamamoto, and Yoshiaki Fukazawa. A metrics suite for measuring reusability of software components. In *Proceedings of the 9th International Symposium on Software Metrics*, pages 211–, Washington, DC, USA, 2003. IEEE Computer Society.

[193] Kurt D Welker. The software maintainability index revisited. *CrossTalk*, pages 18–21, 2001.

[194] Ralf Westphal. Commander im Application Space, Ein Framework zum Umgang mit Parallelverarbeitung. *dotNetPro*, 1:128–135, 2009.

[195] Ralf Westphal. Schnuppern am Application Space. *dotNetPro*, 7:131–136, 2009.

[196] Ralf Westphal. Schnuppern am Application Space, Teil 2. *dotNetPro*, 8:124–129, 2009.

[197] Ralf Westphal. Unendliche Weiten, Komponentenorientiert verteilen mit dem Application Space. *dotNetPro*, 7:124–130, 2009.

[198] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[199] Cathleen Wharton, John Rieman, Clayton Lewis, and Peter Polson. The cognitive walkthrough method: a practitioner's guide. In Jakob Nielsen and Robert L. Mack, editors, *Usability inspection methods*, pages 105–140. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[200] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[201] Dennis Wixon, Karen Holtzblatt, and Stephen Knox. Contextual design: an emergent view of system design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 329–336, New York, NY, USA, 1990. ACM.

[202] Dennis Wixon, Sandra Jones, Linda Tse, and George Casaday. Inspections and design reviews: framework, history and reflection. In Jakob Nielsen and Robert L. Mack, editors, *Usability inspection methods*, pages 77–103. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[203] Dennis R. Wixon, Judy Ramey, Karen Holtzblatt, Hugh Beyer, JoAnn Hackos, Stephanie Rosenbaum, Colleen Page, Sari A. Laakso, and Karri-Pekka Laakso. Usability in practice: field methods evolution and revolution. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '02, pages 880–884, New York, NY, USA, 2002. ACM.

[204] C. Wohlin. Revisiting measurement of software complexity. In *Software Engineering Conference, 1996. Proceedings., 1996 Asia-Pacific*, pages 35–43, 1996.

[205] Fangjun Wu and Tong Yi. A structural complexity metric for software components. In *Proceedings of the The First International Symposium on Data, Privacy, and E-Commerce*, pages 161–163, Washington, DC, USA, 2007. IEEE Computer Society.

[206] Zarnikov, Jan. Energy-efficient Persistence for Extensible Virtual Shared Memory on the Android Operating System. Master's thesis, Vienna University of Technology, 2012.

[207] Hongyu Zhang, Xiuzhen Zhang, and Ming Gu. Predicting defective software components from code complexity measures. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 93–96, Washington, DC, USA, 2007. IEEE Computer Society.

[208] Zhijun Zhang, Victor Basili, and Ben Shneiderman. An empirical study of perspective-based usability inspection. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 42, pages 1346–1350. SAGE Publications, 1998.

[209] Zhijun Zhang, Victor Basili, and Ben Shneiderman. Perspective-based usability inspection: An empirical validation of efficacy. *Empirical Software Engineering*, 4(1):43–69, 1999.

[210] M.F. Zibran. What makes apis difficult to use? *International Journal of Computer Science and Network Security*, 8(4):255–261, 2008.

[211] M.F. Zibran, F.Z. Eishita, and C.K. Roy. Useful, but usable? factors affecting the usability of apis. In *18th Working Conference on Reverse Engineering (WCRE)*, pages 151–155, 2011.

THOMAS SCHELLER

Curriculum Vitae

Personal

Birth date	December 27, 1981
Citizenship	Austria
Family status	married
Address	Lagergasse 31/10, 3425 Langenlebarn
Email	tho.scheller@gmail.com

Education

2002-2007	Bachelor Study “Software & Information Engineering” at Vienna UT Bachelor Thesis: Development of a multiplayer game in Java
2007-2008	Master Study “Software Engineering & Internet Computing” at Vienna UT Projects: • XVSM – Extensible Virtual Shared Memory (Master Thesis) • Genso – Global Education Network for Satellite Operations
2008-2014	Doctoral Study at Vienna UT Research topics: • Application Space: Middleware for distributed and parallel communication • Research on the Usability of APIs • Project AgiLog: Agile middleware for the logistics area

Employment

2003-2014	Software developer at PCSysteme.at GmbH, 1030 Vienna, part time Development of server components for software in the logistics area
2004-2005	Software developer at Juridikum, University of Vienna Development of a book market software
2010-2012	Part-time author for the German developer newspaper VisualStudio1
since 2010	Part-time lecturer at University of Applied Sciences Hagenberg, course for parallel programming and grid computing
since 2014	Software development lead at TouchStar, 1030 Vienna Development of Android apps and mobile tracking solutions for the logistics area

Publications

2011	T. Scheller, E. Kühn: Measurable Concepts for the Usability of Software Components, 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2011.
2012	T. Scheller, E. Kühn: Influencing Factors on the Usability of API Classes and Methods, 19th Annual IEEE International Conference and Workshops on the Engineering of Computer Based Systems, IEEE, 2012.
	E. Kühn, A. Marek, T. Scheller, V. Sesum-Cavic, M. Vögler: A Space-Based Generic Pattern for Self-Initiative Load Clustering Agents, 14th International Conference on Coordination Models and Languages (COORDINATION), Springer, LNCS, 2012.
2013	T. Scheller and E. Kühn: Influence of code completion methods on API usability: A case study, in The 12th IASTED International Conference on Software Engineering, ActaPress, 2013.
	T. Scheller and E. Kühn: Usability evaluation of configuration-based API design concepts, in International Conference on Human Factors in Computing & Informatics, Springer Berlin Heidelberg, 2013.
	E. Kühn, S. Craß, G. Joskowicz, A. Marek, and T. Scheller, Peer-based Programming Model for Coordination Patterns, 15th International Conference on Coordination Models and Languages (COORDINATION), Springer, LNCS, 2013.
2014	T. Scheller and E. Kühn: Automated Measurement of API Usability: The API Concepts Framework, to be published in Information and Software Technology, Elsevier, 2014.