FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Active Objects Revisited:

A Concurrency Library for Java Based on an Object-Oriented Approach to Parallelism

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Philipp Grandits
Matrikelnummer 0826361

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

Wien, 2.10.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Active Objects Revisited:

A Concurrency Library for Java Based on an Object-Oriented Approach to Parallelism

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Philipp Grandits
Registration Number 0826361

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

Vienna, 2.10.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Philipp Grandits
Schulgasse 24, 7411 Markt Allhau

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

I would like to express my appreciation to Professor Franz Puntigam for supervising my thesis and giving me the opportunity to address this interesting research topic. He provided me with valuable advice and ideas in the various phases of the project and helped me to view the subject matter from different perspectives.

I also wish to thank the participants of the associated seminar for interesting discussions and useful feedback about my work.

I would like to express my sincere gratitude to my family and friends for supporting me throughout my studies and this research project. I want to especially thank my friend Karin for her constant encouragement and understanding when it was most required. I would also like to thank my parents for their moral and financial support throughout these years, which provided the basis of this thesis.

Abstract

In recent years, parallelism and concurrency have become significant topics because applications need adequate language constructs in order to completely exploit the performance offered by multi-core processors and multi-processor systems. As a result, Erlang regained importance due to its efficient implementation of the actor model. In this technique, components use an independent thread to handle incoming messages. A related approach is the active object pattern. It expands on the idea of actors, but operates on method requests rather than messages. These kinds of instances are automatically generated whenever one of the active object's methods is called by an external thread. This strategy decouples the invocation of an operation from its execution. The incoming requests are scheduled one after the other to guarantee that only a single method is performed at a time. The concept facilitates software development because synchronisation concerns are alleviated. However, there does not currently exist a Java implementation of active objects that integrates appropriately into the language.

Therefore, this thesis addresses the question which functions and features are reasonable for such a realisation of the pattern, and how they can be incorporated into a library for Java through reflective programming and bytecode instrumentation.

The first step to achieve this objective is to study associated literature from the initial surge of research, as well as more modern work that revisited the methodology after the original interest declined. Certain advanced features are selected from the gathered information. They are then cast into a model together with the basic functionality. The design is realised by a proof of concept implementation. The resulting tool can be used to develop classes of active objects. Lastly, the suitability of the chosen constructs is evaluated. For this purpose, a concurrent scenario is programmed both conventionally and with the aid of the library. These applications are used for determining specific code and performance characteristics of the individual approaches. Thereafter, the collected data are compared and analysed.

The evaluation shows that the presented mechanisms ease the development of parallel software because the focus is moved from the synchronisation to the actual core functionality of the components. This practice reduces the necessary implementation effort and helps to avoid faults in the end product. However, in some categories the analysis reveals inferior efficiency of the active object applications when matched against the traditional style. Although the difference is not overwhelming, it may be a deciding factor in some cases. The technique can become more appealing by closing the performance gap through further optimisations and extensions. All things considered, the outcome indicates that a well-integrated active object feature is a competitive approach if compared to standard Java concurrency.

Kurzfassung

In jüngster Vergangenheit wurden Parallelisierung und Nebenläufigkeit zu einem vieldiskutierten Forschungsgebiet, da für die Programmierung von Anwendungen geeignete Sprachkonstrukte benötigt werden, um die Leistung von Mehrkernprozessoren und Mehrprozessorsystemen vollständig auszunutzen. Aus diesem Grund hat die Programmiersprache Erlang an Wichtigkeit gewonnen, da sie eine effiziente Implementierung des Aktorenmodells bereitstellt. Dieses stützt sich auf Aktoren, die einen eigenen Thread verwenden, um eintreffende Nachrichten zu verarbeiten. Ein verwandter Ansatz, der dieses Konzept erweitert, ist das Active-Object-Pattern. Anstatt von Nachrichten operiert das Modell auf sogenannten Method-Requests. Diese werden automatisch generiert, wenn eine der Methoden des aktiven Objekts durch einen externen Thread aufgerufen wird und dienen dazu, den Aufruf von der Ausführung zu entkoppeln. Die hereinkommenden Method-Requests werden nacheinander abgearbeitet um sicherzustellen, dass zu jedem Zeitpunkt höchstens eine einzige Methode ausgeführt wird. Dieses Konzept unterstützt die Entwicklung, weil Synchronisierungsaufgaben erleichtert werden. Allerdings existiert derzeit keine Java-Implementierung, die dieses Verfahren angemessen in die Sprache integriert.

Deshalb behandelt diese Diplomarbeit die Frage, welche Funktionen und Features für eine Realisierung des Active-Object-Patterns sinnvoll sind und wie diese durch reflektive Programmierung und Bytecode-Instrumentation in eine Bibliothek für Java eingebunden werden können.

Zur Erreichung dieses Ziels wird zuerst die zugehörige Literatur aus den Anfängen der Forschung in diesem Gebiet genauer betrachtet. Außerdem werden auch aktuellere Arbeiten untersucht, die diese Methodologie wieder aufgriffen, nachdem das anfängliche Interesse schwand. Dabei werden bestimmte erweiterte Features aus der gesammelten Information ausgewählt, die anschließend zusammen mit der Basisfunktionalität modelliert werden. Dieses Design wird durch eine Proof-of-Concept-Implementierung realisiert, die verwendet werden kann, um Applikationen basierend auf aktiven Objekten zu entwickeln. Letztendlich wird die Eignung der gewählten Konzepte evaluiert. Dazu wird ein nebenläufiges Szenario sowohl konventionell als auch mit Hilfe der Funktionalität der Bibliothek programmiert, um spezifische Leistungs- und Implementierungscharakteristika der verschiedenen Anwendungen zu bestimmen. Anschließend werden die gesammelten Ergebnisse verglichen und analysiert.

Die Evaluierung zeigt, dass die Mechanismen der Bibliothek die Entwicklung eines parallelen Programms erleichtern, weil diese den Fokus von der Synchronisation auf die eigentliche Kernfunktionalität der Komponenten lenken. Dieser Umstand reduziert den notwendigen Entwicklungsaufwand und hilft, Fehler im Softwareprodukt zu vermeiden. Allerdings zeigt die Analyse, dass der Ansatz in manchen Kategorien auch eine schlechtere Performanz mit sich bringt, wenn man aktive Objekte dem herkömmlichen Stil gegenüberstellt. Auch wenn der Unterschied

nicht überwältigend ist, kann dieser ein entscheidender Faktor in manchen Fällen sein. Durch das Schließen des Leistungsunterschieds im Zuge weiterer Optimierungen und Erweiterungen würde die Verwendung des vorgeschlagenen Programmiermodells reizvoller werden. Alles in allem deutet das Ergebnis darauf hin, dass eine gut integrierte Implementierung von aktiven Objekten im Vergleich zu normalen Java-Nebenläufigkeitskonstrukten ein konkurrenzfähiger Ansatz ist.

Contents

CHAPTER 1

Introduction

1.1 Motivation

In the last decade, multi-core processors and multi-processor systems have been on the advance because physical constraints limit the performance enhancement of individual cores [16, 17]. In order to further increase the computational power, hardware manufacturers began embedding a number of cores on one chip and providing means for the installation of several CPUs inside a single system.

This technological shift enforces new demands on development tools and practices. To utilise the full potential of modern processors, software has to be written in a concurrent fashion. For this reason, adequate language features are necessary, which enable an efficient translation of functional requirements into maintainable applications with a high degree of parallelism [17]. However, most programming languages have not made significant progress in this regard as the technology has advanced. Conventional models employ shared-state concurrency (cf. [7, 18]), which means different threads jointly access the same state of variables and objects. To achieve coordination of the activities, concepts like semaphores and monitors are used. These constructs allow the programmer to define the exact position in the code where synchronisation and the creation of new threads take place, granting full control over the aspects of parallelism. The downside of this approach is that the development of concurrent scenarios becomes complex and error-prone [7,32,53]. With massive multi-core architectures, also known as many-core architectures, on the horizon, there is little prospect of improvement in this respect [51]. These systems accommodate tens to hundreds or even thousands of cores, thus supporting a vast amount of simultaneous threads. An alternate programming model was utilised by the creators of Erlang and has been regularly referred to in this context [33, 51]. They settled for independent actors that do not share a common state and communicate among each other through the use of messages [6]. Every actor has its own private thread for handling messages consecutively. This notion implies that only a single thread is accessing and manipulating the internal state at any given time. Hence, exclusive control over the actor can be presumed while processing a message. This restriction eases the development of individual components significantly. The model was

transferred to various other languages like Scala or Groovy, either directly on the language-level or by means of libraries. Implementations for Java do also exist, but its object-oriented nature does not seem to match the concept because actors use messages as the primary mechanism of communication.

Another related approach alleviates this problem by passing method invocation requests instead of messages. It is called the active object pattern [31]. Similar to an actor, an active object possesses a single thread responsible for executing the method invocation requests inside the instance's boundaries. Because their use allows tighter integration into existing object-oriented idioms, this practice appears to be better suited to Java.

1.2 Problem Statement

Although the underlying design pattern can be explicitly used, there is no native support for active objects in Java as of yet. Furthermore, no framework is available that incorporates them adequately into the language. Therefore, a library implementation is required to enable the utilisation of these constructs in multi-threaded applications. Reflective programming and bytecode instrumentation are valuable techniques to achieve the tight language integration necessary to facilitate the effective and efficient usage of the model.

Thus, the objective of this thesis is to answer the following questions:

- Which functions and features are useful and sensible in an active object implementation to enable effective development of concurrent scenarios?
- How can the identified functions and features be realised in a library for Java by means of reflection and bytecode instrumentation?

1.3 Aim of the Work

The goal is to provide mechanisms to be able to use active objects as elements in applications. This thesis comprises the detailed description and realisation of selected functions and features. The latter are identified by review of associated literature. The discovered techniques are analysed and compared in terms of their suitability for a Java library. To accomplish this functionality, the corresponding capabilities and aspects are divided into a number of components.

The findings include the base model of the library as well as an example application demonstrating the manner of utilisation. To that end, an extended producer-consumer situation is implemented, a problem regularly used to illustrate concurrency. The core components and their interactions are documented in the thesis and in the code via JavaDoc. Additionally, the functionality is validated through appropriate unit test cases. To give insight into the internal operating principle of the library, various aspects of the implementation are covered explicitly. For this purpose, code samples are presented and explained in detail.

1.4 Methodological Approach

The methodological approach is structured into the following steps:

1. *Additional literature investigation.* Further information is collected about existing tools and languages that cover concurrency and asynchronous computations. The focus lies primarily on the active object pattern, the actor model and related realisations. Nevertheless, other concepts are also being explored to examine the current circumstances from varied perspectives.

2. *Analysis of collected information.* The previously gathered data is set in context of an implementation in a Java library. Since the syntactic and semantic checks of the compiler restrict the possible representation of a particular function, the viability of different strategies needs to be compared. A potential problem is that certain aspects cannot be implemented in a sensible and usable way. Technical limitations may be another reason preventing a feature from being integrated.

3. *Modelling of the components.* The main objective of this step is to create a detailed description of the way in which the desired features are accomplished. Therefore, the functionality needs to be divided into individual components with definite responsibilities. This separation enables designing the functionality and the relationships of the entities. The internals specified in this phase include the behaviour of the (interface) methods and important instance variables.

4. *Implementation of the components.* In this stage, the aforementioned elements are implemented for Java 8 using a test-driven development procedure. It is based on unit tests written in JUnit 4. The verification process is not in the scope of this thesis. At the same time, the discrete components are gradually combined with each other. If possible, the integration is validated by unit testing as well.

5. *Implementation of an example application.* A parallel producer-consumer scenario is programmed in Java 8 to illustrate the library's functionality and usage. Additionally, this use case is realised by means of standard concurrency constructs to obtain a baseline for the analysis. Unit testing of the example is not intended.

6. *Definition of evaluation criteria.* This part addresses the measurement categories used for the comparison of the applications. These classes comprise general performance characteristics, scenario-specific metrics and the inspection of source code. The settings of any parameters are also fixed at this point.

7. *Collection of results and analysis.* Lastly, the written programs are assessed on the basis of the previously determined criteria. Thereby, the analysis puts the gathered metrics and characteristics of the active object approaches into perspective of the conventional implementation. These findings are utilised to determine if the goals of the thesis have been met and to identify potential future work.

1.5 Structure of the Master's Thesis

This section outlines the layout of the main part of this thesis. Firstly, Chapter 1 introduces the topic and addresses the thesis's objectives as well as the manner in which they are achieved. Chapter 2 continues by providing an overview of the associated literature and giving arguments for the work's relevance in the context of already existing research. Result handling strategies are treated in Chapter 3, where they are also inspected in terms of their suitability for an inclusion in the design. Chapter 4 explains the proposed model for the automated incorporation of active objects in greater detail. Its proof of concept implementation is then subject of Chapter 5, which describes the general bytecode instrumentation mechanisms and the structure of the library. Chapter 6 outlines the criteria and the concurrent scenario used for the evaluation. The collected results are then presented and examined in Chapter 7. This part does also contain a comparison of the different approaches and tries to deduct more general statements from the findings. Finally, Chapter 8 summarises the main efforts and results, and gives an outlook of future work with regard to the active object pattern. Furthermore, the individual entities of the evaluation scenario are characterised in Appendix A.

CHAPTER 2

State of the Art

Over the course of several decades, the actor model and active objects have received several important research articles. This chapter presents and examines related studies. It begins in Section 2.1 by taking a glimpse at the actor model and its features. In Section 2.2, the active object pattern is explained in detail and put in comparison to the actor model. Section 2.3 covers more advanced topics regarding active objects, for instance coboxes. Lastly, Section 2.4 gives insight into the relevance of this master's thesis and contrasts it to the previously mentioned studies.

2.1 Actor Model

At this point, we examine the origins of the active object pattern to gain a better understanding of its fundamental idea and core concepts. The research on this matter dates back to actors. This section describes the substantial mechanisms and advanced realisation properties of the model.

Fundamental Characteristics

Agha and Hewitt [5, 6] specify the basic constructs and describe an actor as a *"computational agent which carries out its actions in response to processing a communication"*. The model was proposed to deal with problems that arise when shared objects have a mutable state and is intended to support two noteworthy aspects:

- *Inherent Parallelism* implies that there is no hidden parallelism, but instead, the structure of the application's code reveals the actual amount of concurrency.
- *Dynamic reconfigurability* means the programming model enables the developed systems to easily accommodate new objects.

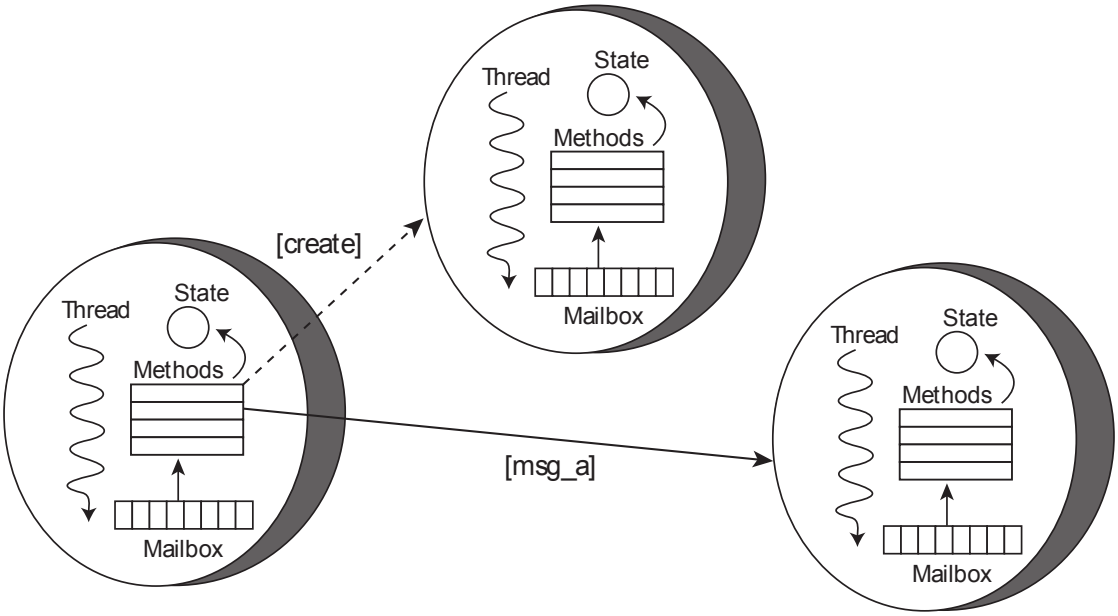

Figure 2.1: An overview of the components of actors and the interactions between them. [27]

In addition to the usual program constructs, actors interact among themselves in several different ways. Most importantly, they communicate asynchronously via message-passing. Therefore, every actor has its own mailbox which stores the incoming correspondence. The respective address is sent to another actor within a message as any other object. Furthermore, an actor can create additional actors, for example upon arrival of a particular notification. An interesting feature is the declaration of a so-called *replacement actor*. It functions as a substitute and handles messages in place of the original actor. The propagation of the mailbox's address of a newly created actor and the specification of a replacement actor contribute to the reconfigurability of the programming model.

Realisation Properties

Although actors are characterised to be inherently concurrent, the description in [6] does not specify the way this capability is achieved. Implementations and other research papers concerned with the topic show that a thread is responsible for processing the received messages [27]. At any given time, only one thread can be active inside an actor's body so as to handle the communication and manipulate the internal state. Figure 2.1 visualises the interactions between actors and their components. Karmani et al. [27] list several important properties that should be supported by an implementation of this concept.

Encapsulation. The internal state is protected from illegal tampering. This goal is achieved by restricting the access to certain aspects of the actor. The model rests on two important qualities in this context:

- *State encapsulation* specifies that one actor can only manipulate the state of another actor by sending messages. Direct access is prohibited.
- *Safe messaging* further braces the decoupling by prescribing that the contents of messages are passed by value (cf. call-by-value semantics). This restriction is intended to prevent unwanted or unexpected state sharing.

Fair Scheduling. The underlying scheduling techniques treat actors equally. If this property is not guaranteed, severe repercussions are possible. These problems include the starvation of actors, a loss of performance or the hindrance to progress the application's tasks (cf. live lock). A distinction is made between two related ideas:

- *Fair scheduling of actors* implies an actor will eventually be allowed to receive and process its incoming messages.
- *Fair scheduling of messages* means a sent message will ultimately be received and processed by an actor.

Location Transparency. The naming of actors is independent of their address space. The main objective is to improve the flexibility through the introduction of an additional layer between them and their clients. Thereby, the library is responsible for locating actors in the implemented system on behalf of the developer. Finding distributed actors in the network is a possible consequence of this mechanism. This property aids in preventing direct access of the internal state.

Mobility. An actor can be migrated to another node in the network. This relocation could be necessary to improve the performance of the application through load-balancing. As a result, the system is required to move the actor and the concerned components to the target machine. Another vital aspect is that the other actors need to be notified of this change. Thus, the model must consider that invocations of local actors can result in remote calls in the future.

Examples of implementations in Java are Kilim and ActorFoundry. A comparison between different libraries for the Java Virtual Machine is the subject of [27].

2.2 Active Object Pattern

This section is concerned with the pattern as originally proposed by Lavender and Schmidt in [31] and the comparison to the previously introduced actors. Note though that the concept of active objects has a long history which must not be forgotten. They were extensively explained by Nierstrasz in [35] and the underlying type theory was discussed in [37, 38]. Futhermore, Nierstrasz et al. designed and implemented Hybrid, a programming language based on active objects [28, 34, 36]. Most of these works address the topic at the language-level. We have therefore chosen the paper of Lavender and Schmidt as the starting point because it is better suited for a library implementation. The base pattern is described as a way to *"decouple method*

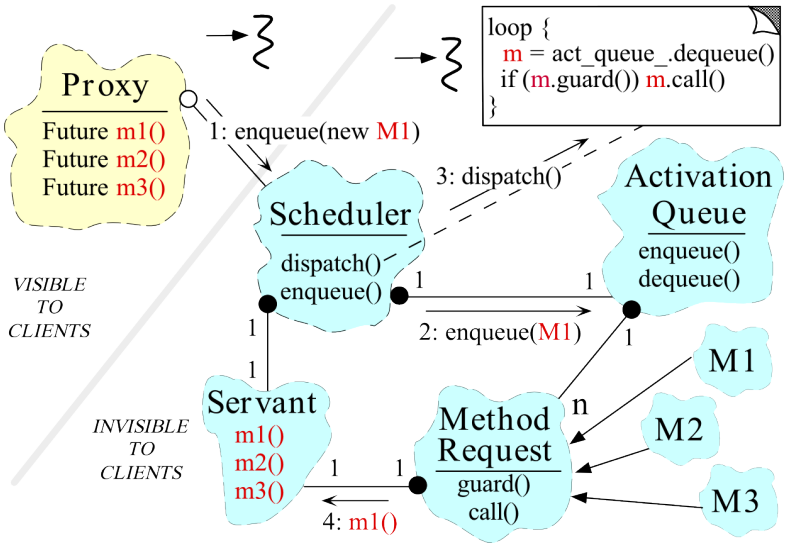

Figure 2.2: The original booch class diagram that shows the key components of the active object pattern. [31]

execution from method invocation in order to simplify synchronised access . . . " [31]. We now turn towards the active object's components, the stages of an active call and the comparison to the previously mentioned actor properties.

Core Components

Each active object is thought of as an autonomous actor. It consists of several internal components with certain responsibilities. The basic structure of the model is depicted in Figure 2.2. There are six essential entities in the original pattern.

Proxy. The *proxy* is the active object's connection to the outside world. It provides a publicly available interface to the clients. While conventional methods directly perform the realised procedure, a *proxy method* has to transparently construct a method request and enqueue this instance into the activation queue of the scheduler. Additionally, a future is returned to the caller which grants access to the invocation result. The proxy method is executed by the client thread.

Scheduler. The main job of the *scheduler* is to execute the method requests created by the proxy methods. For this purpose, it holds an activation queue which is used to facilitate the management of pending requests. To achieve the required degree of decoupling, the scheduler runs in a different thread than the clients. A concrete implementation can incorporate a variety of factors in its scheduling decision, for example the temporal ordering of arrival.

Activation queue. The *activation queue* is the link between the proxy and the scheduler. This collection holds the method requests created by the proxy. They are stored until the scheduler

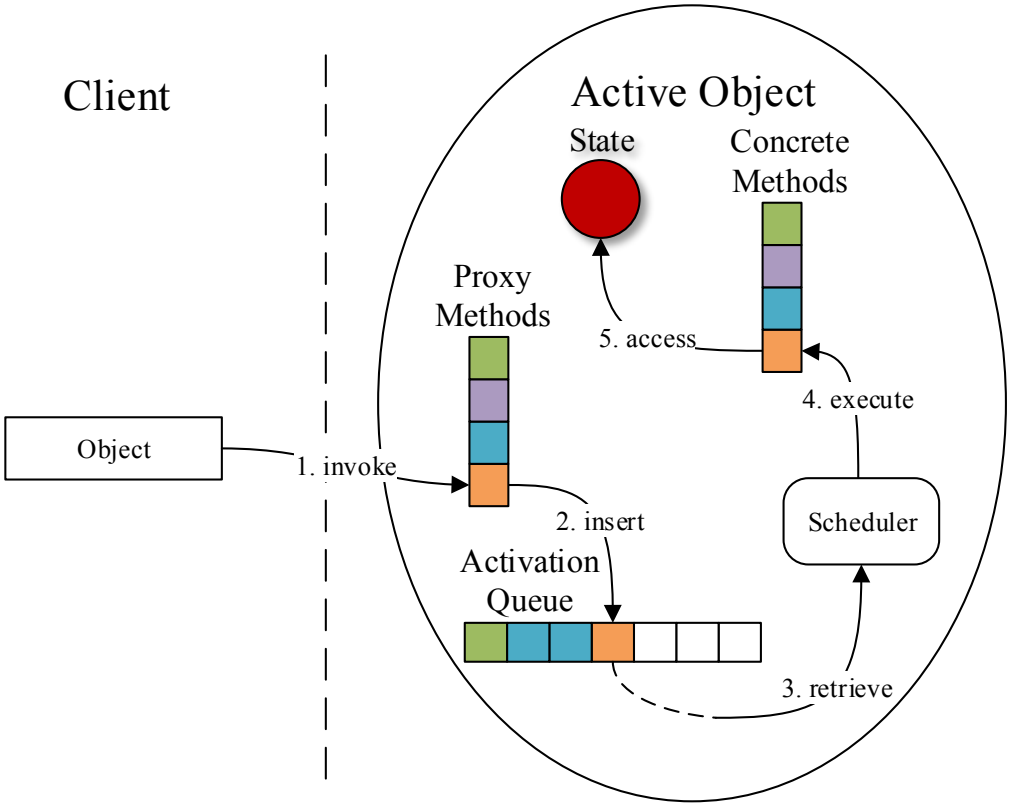

Figure 2.3: The control flow of a method call inside an active object.

claims them. The order of retrieval depends on the implementation. To cover a variety of use cases, the library may offer different alternatives, for example realisations based on a FIFO (first-in, first-out) or a priority-driven strategy.

Method request. A *method request* contains the necessary details in order to invoke a servant method. The stored information includes the concrete operation and the parameters of the call. The interface in Figure 2.2 does also specify a `guard` function. It is used to check if the predetermined execution conditions are met. Method requests are a key component in the active object pattern. They are instantiated by the proxy methods and then appended to the activation queue. There they remain until the scheduler fetches and performs the individual instances. These events are illustrated in the steps two, three and four of Figure 2.3. The outlined approach enables the separation of the invocation in the client thread from the actual execution in the scheduler thread.

Servant. The *servant* of the active object implements the core features clients want to utilise. Its methods are allowed to access and manipulate the state of the active object without any restrictions. Whenever the scheduler conducts the execution of a method request, an operation

of the servant is implicitly invoked. Thus, the procedures of this entity run in the same thread as the scheduler.

Future. *Futures* act as placeholders for the result of asynchronous computations [10]. These kinds of objects are instantiated in the process of creating and queueing a method request. They are returned by the proxy method to enable client-side access to the output of the decoupled method invocation. Among others, a future usually provides functions for querying the execution state and obtaining the result, either by polling or blocked waiting.

Active Invocation

The crucial characteristic of the pattern is that the proxy runs in the client's thread, whereas the scheduler and servant both share a thread that is separate from the caller. According to the description in [31], an interaction between different active objects can be divided into three phases. This partitioning is visualised in Figure 2.4. The sequence below outlines the procedure:

1. *Method request construction.* As soon as a proxy method is invoked, it creates a new method request. This instance is then transmitted to the scheduler before returning the associated future. The control is immediately handed back to the caller and the execution continues after the statement that concerned the proxy method.

2. *Scheduling and execution of the request.* In the next step, the scheduler inserts the received request into its activation queue. At some point the scheduling thread dequeues the same instance. Before carrying out the request, its `guard` function is examined. If this condition is fulfilled, the mechanism instructs the execution. This action starts the actual servant method. Otherwise, the request is inserted back into the queue.

3. *Completion of the invocation.* Once the servant method terminates, the result is set inside the future and the active invocation reaches its concluding stage. At this point, the client's thread may retrieve the final value from the future returned in the first phase.

Actor Properties

Since the active object pattern can be seen as an object-oriented alternative to the actor model, they fulfil similar purposes. Therefore, the properties mentioned in Section 2.1 are relevant to this concept. The following enumeration is devoted to the analysis of the individual characteristics and their realisation in the context of the active object pattern:

Encapsulation. As previously explained, *encapsulation* is divided into two aspects:

- *State encapsulation* is achieved through the pattern itself. An implementation may allow the definition of both active and passive methods in the same class. The latter type is directly executed as any conventional method. The programmer is responsible for protecting the internal state from external access. This safeguard can be provided by defining

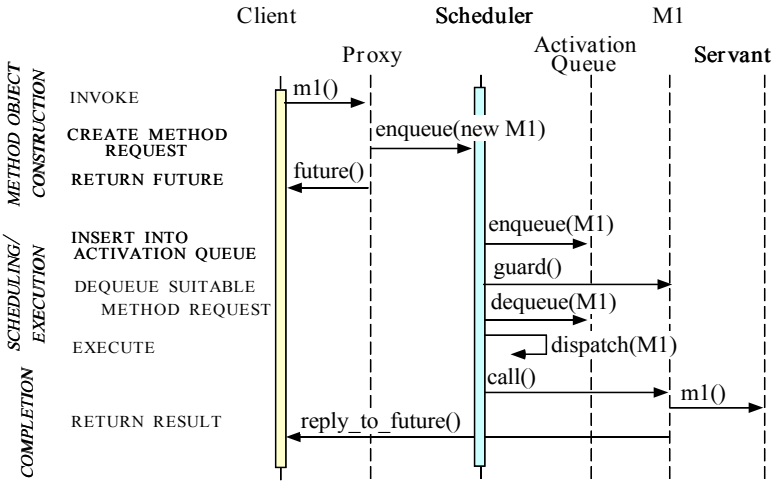

Figure 2.4: Sequence diagram of the different phases of an active invocation. [31]

only active methods or by restricting the conventional type to operations which return immutable[1] objects (cf. getter method of a final instance variable in Java).

- *Safe messaging* requires parameters to be passed by value. Although Java relies on call-by-value semantics for all types, in reality only primitives are deeply copied. In the case of non-primitive types, the reference is simply duplicated and used as the parameter of the call. This shortcoming may cause unexpected state sharing, for example if a mutable object is sent to another active object. Because a language like Java is not restricted to certain concurrency constructs, a library implementation of the pattern would have difficulties in distinguishing intended from erroneous method calls. Therefore, the application's developer has to guarantee safe messaging by manually copying objects if necessary.

Fair scheduling. *Fair scheduling* depends on the realisation as well as the language and operating system:

- *Active object fairness* is determined by the chosen threading model. For example, if an implementation maps an active object to a JVM thread, the library is as fair as the underlying JVM or operating system scheduler (cf. actor fairness [27]).

- *Method request fairness* depends solely on the order in which pending method requests are selected for processing from the activation queue. However, up to this point no effort was made to specify a suitable queueing strategy. In the original actor model, message fairness is described as the concept that *"a message is eventually delivered to its destination actor, unless the destination actor is permanently disabled (in an infinite loop or trying to do*

[1] Immutable objects do not change their state throughout their lifetime. Once such an instance is created, it remains stable.

an illegal operation)" [31]. Under the assumption that no incoming method requests are prioritised, the first-in first-out (FIFO) strategy fulfils this requirement.

Location transparency. *Location transparency* has to be supported by the library implementation. This feature prescribes that active objects have to be decoupled from references, pointers and addresses which are internally used by the programming language. Furthermore, the model enables a high degree of distribution. Active objects may be spread across the network to improve performance. For this purpose, the library would have to provide some sort of lookup mechanism. Instead of passing references to other classes, the active objects can be located by this method. This technique is especially interesting when instances communicate over the network. Hence, method invocations on an active object would have to be decoupled from its physical location. This task can be fulfilled by checking the network location of the object before a call is made. Another possibility is to replace actual references with a smart pointer. It comprises the whereabouts of the real active object and the logic for forwarding the call. If the object is moved to a different node, the container is notified and automatically updated. Thereby, multiple clients share the same immutable instance for a given active object in order to reduce the overhead and the memory consumption.

Mobility. The *mobility* property facilitates the migration of an active object to other network nodes. The system may perform this relocation automatically or provide language constructs which enable developers to conduct a transfer themselves. A particular reason for an automatic migration may be to reduce the duration of active calls because the corresponding active object is primarily used by clients on another node. An implementation has to consider the current state of an active object when a transfer is imminent. If the activation queue is empty and the scheduler does not momentarily process any method requests, the system only has to transmit the state of the object. The remaining components can be newly created on the target machine. On the contrary, migration is more complex if the activation queue is filled and the scheduler is currently processing method requests. In this case, the state of the active object and the complete activation queue have to be copied to the new node. The system also needs to coordinate the migration with the scheduler, so that no requests are executed at the time of relocation. This precaution ensures the state of the object is stable when it is sent over the network.

In this thesis, we will only incorporate state encapsulation and both forms of fair scheduling into the realisation. A satisfactory solution to safe messaging in Java is deemed a delicate matter with many challenges. A secondary treatment would simply not do justice to the subject and the encountered problems. Moreover, location transparency and mobility are left open as extensions for future work.

The concept of active objects has several related research papers. [48] describes its use as part of a speech recognition framework. Similarly, [47] utilises the pattern to enhance the concurrency of object request broker middlewares. Furthermore, in [20] an autonomous agent architecture is introduced, which may be constructed from active objects.

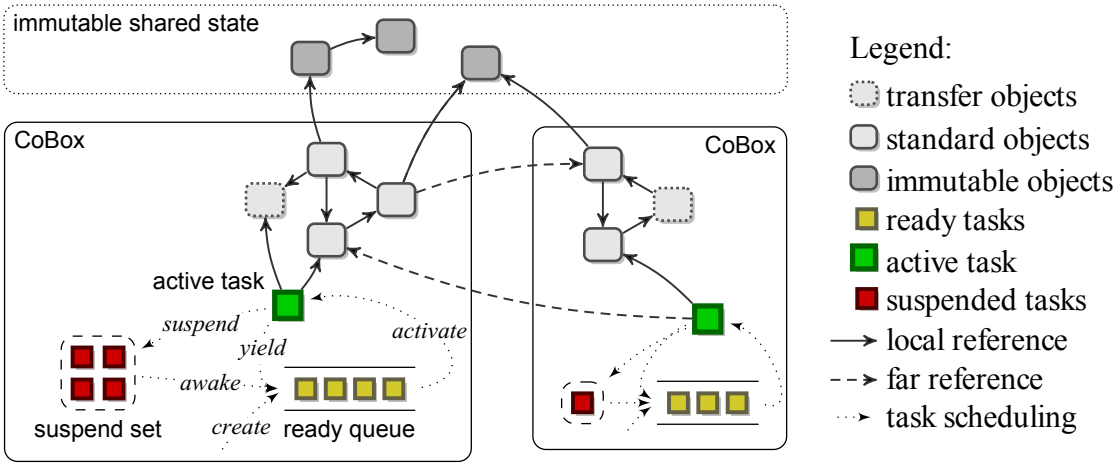

Figure 2.5: The various components of the JCobox model. [45]

2.3 Advanced Techniques Built Upon Active Objects

This section discusses related research work that expands on the idea of active objects.

The Cobox Programming Model

A cobox is a conceptual container holding multiple objects and queues of tasks. It resembles an active object in the sense that both entities are concurrently running components and they encapsulate the state of the embodied system part. This model was proposed by Schäfer and Poetzsch-Heffter [44–46]. Its three main concepts are task states, object types and the distinction between different kinds of references. These ideas are also illustrated in Figure 2.5. The library implementation named *JCobox* is an extension of Java. It uses the compiler frontend Polyglot [4] to enhance the language with an operator for asynchronous calls and introduces an additional kind of `new` expression. Furthermore, a set of annotations and methods are provided for different purposes. The next paragraphs are devoted to individual concepts and the library implementation.

Tasks and Task States

A *task* represents an obligation of a cobox to conduct a certain activity. They are similar to method requests but expand on the mechanism in various ways. When an operation is addressed from outside the cobox of the targeted object, the invocation is performed asynchronously and a task is created implicitly. In contrast, a call within the corresponding cobox is directly executed. A crucial characteristic of the model are the three states a task can be in:

- The *ready state* concerns tasks that are prepared to being run. They are collected in the *ready queue*, which is based on the first-in first-out (FIFO) strategy. Newly created tasks

```
1 @CoBox class AServer implements Server {
2    List<Session> sessions = new ArrayList<Session>();
3    Session connect(Client c) { return new ASession(c); }
4    void broadcast(Msg m) { ... }
5    class ASession implements Session { ... }
6 }
```

Algorithm 2.1: The class definition of the cobox type `AServer`. [45]

also fall under this category. As soon as the active task releases control, the next task of the ready queue is scheduled.

- At any given time, only one task can be in the *running state*. This is referred to as the *active task*. It is executed inside the cobox and has exclusive access to the internal state. The active task returns control to the system upon termination, suspension or by explicitly *yielding*. The latter possibility is a special feature. It appends the active task to the ready queue.
- The active task reaches the *suspended state* when waiting for the fulfilment of a condition. For instance, this concerns *awaiting* the completion of a future. Such tasks are stored inside the *suspended set* and are awakened once the respective condition is satisfied. This revival is achieved by their removal from the suspended set and insertion into the ready queue.

The programming model enables interleaved control flows through a set of predefined constructs like the aforementioned await and yield. The idea of scheduling threads in a cooperative manner is based on Creol [15, 25].

Reference Types

Since the objects of a cobox can be shared with other coboxes, a distinction is made between different types of references:

- A *local reference* provides a connection to an object contained within the same cobox. This kind of instance is also called a *local object*. Such references enable direct field access. Furthermore, they can be used to immediately execute methods of the target object.
- A *far reference* points to an object outside of the current cobox. This sort of link does not allow direct access to fields. Furthermore, method calls are only performed asynchronously. Because they cannot deliver the actual result instantly, a future is returned.

Object Types

The model supports three concrete forms of objects. They are distinguished in the way they are shared between coboxes. The individual types are explained below.

Firstly, *standard objects* are instances of either plain classes or cobox types. With respect to JCobox, the latter are declared using the `@CoBox` annotation. Algorithm 2.1 shows an exemplary definition. On the other hand, a plain class does not exhibit a particular annotation. The characteristics of these instances are summarised in the ensuing list.

- Both kinds of standard objects are passed by reference.
- If a cobox does not own a standard object, the far reference semantics apply.
- A cobox class can be instantiated by a conventional `new` expression. This construction always leads to the creation of a new cobox, even if the instantiation is performed by a task of another cobox.
- By default, an object of a non-cobox class is created inside the cobox of the producing task.
- The extended `new` expression enables the declaration of a target cobox. The statement in the example below creates a `Session` instance inside the cobox that holds the object referenced by `server2`:
  ```
  new Session([...]) in(server2)
  ```

Secondly, *transfer objects* are used to pass data to another cobox when dependency through a far reference has to be avoided. JCobox provides the annotation `@Transfer` to denote these kinds of classes. The points hereafter outline the properties of this type.

- Transfer objects employ the local reference behaviour.
- Whenever such an object is passed, a deep copy of it is provided to the destination cobox. This instance is treated as a local object.
- Two coboxes cannot share the same transfer object reference.
- The following code sample illustrates the definition of a transfer class for messages:
  ```
  @Transfer class Msg { String user; String content; }
  ```

Thirdly, *immutable objects* are an alternative to the possibly inefficient transfer objects. For this purpose, the concerned classes are marked with the `@Immutable` annotation. Their properties are described below.

- Immutable objects are passed by reference.
- Although instances of this type do not belong to a specific cobox, they are treated as local objects and have the semantics of local references.
- Because immutable objects do not change their state, they can be shared and accessed safely by concurrently running threads.

- An immutable object can contain references to other immutable objects or to standard objects. The former are treated as local references, while the latter utilise far reference semantics. References to transfer objects cannot be used inside of immutable objects.

Call semantics

JCobox extends Java by two mechanisms for performing asynchronous calls on objects. They differ merely in a single aspect. The `!` operator preserves the partial order[2] of invocations, whereas the `!!` operator does not necessarily do so. The library only allows methods of standard objects to be called asynchronously. In comparison, methods of immutable and transfer objects always have to be invoked directly. The usage of the operators is outlined in the next segment:

```
Fut<Session> result = server ! connect(this);
result = server !! connect(this);
```

Cooperative scheduling

In JCobox, futures are represented by the generic interface `Fut<V>` with the type parameter `V`. In this context, `V` relates to the return type of the associated method. `Fut` offers two functions for retrieving the result value. They are different in terms of the cooperative usage of the underlying thread:

- `get()` blocks the thread and does not allow any tasks to be run while waiting for the result.
- Calling `await()` on an incomplete future adds the active task to the suspended set. Once the future is completed, the task is awakened and appended to the ready queue.

As a result, the `await()` function enables other tasks to be processed in the meantime, while `get()` does not. This cooperative approach can only be used if the corresponding task is able to establish the invariants of the cobox before the invocation is performed. Invoking the static method `JCoBox.yield()` is another way to return control to the system. This procedure causes the active task to be appended to the ready queue. The variant `JCoBox.yield(long waitTime, TimeUnit unit)` supports the specification of a time frame which needs to elapse before the task is inserted back into the ready queue.

Application-Level Scheduling in Creol

The researchers of Creol have addressed the question how priorities can be specified at the application-level for active objects [39]. Algorithm 2.2 defines three types. They are intended to recreate a mutex. Lines 1 to 5 illustrate an interface named `PrioritizedResource`. It declares two operations for managing the ownership of the underlying resource and a *priority range* (line 2) for client-side calls. Every invocation of the associated methods can be complemented with a priority in the following way:

[2] Under the assumption that a call x is performed prior to a call y and that x and y have the same source and destination cobox, the partial order is preserved if x is always executed before y.

```
1  interface PrioritizedResource begin
2     priority range 0..9
3     op request()
4     op release()
5  end
6  interface Resource begin
7     op request()
8     op release()
9  end
10 class ExclusiveResource implements Resource begin
11    var taken := false;
12    priority range 0..2
13    op request() priority(2) ==
14       await ~taken priority(1);
15       taken := true;
16    op release() priority(0) ==
17       taken := false;
18 end
```

Algorithm 2.2: An example of application-level scheduling. [39]

```
resource ! request() priority(3);
```

If the statement omits the trailing `priority` expression, a default value is used. This direct mode is only one possibility for controlling the precedence in an application. Lines 6 to 13 demonstrate the definition of an interface with the name `Resource`. Because it does not specify a priority range, an implementation may do so in its body. Shown from line 10 to 18 is the class `ExclusiveResource`, a concrete realisation of `Resource`. Imagine that invocations of `release` experience starvation in the concerned program because `request` is constantly called. In this case, we have to define a range in line 12 and assign priorities for both operations in line 13 and 16. Since by default a lower number indicates higher precedence, `release` is given preference over `request`.

Moreover, the declaration of priorities in Creol is not restricted to constants. A developer can use expressions which resolve to discrete values at runtime. The only constraint is that the evaluated result has to be inside the given boundaries.

Multi-Threaded Active Objects

In the works of Henrio et al. [21–23], effort is made to provide means by which several threads can reside inside a single active object and process method requests in parallel. These so-called *multi-active objects* are intended to improve the performance, reduce the response time of invocations and prevent deadlocks. The concept is realised by declaring groups, categorising the methods and defining compatibility constraints. The latter determine which groups of requests can run simultaneously. A particular group can also be compatible with itself. The rest of this subsection addresses the proposed features in greater detail and expresses criticism of the chosen techniques.

```
1  @DefineGroups({
2     @Group(name="join", selfCompatible=false)
3     @Group(name="routing", selfCompatible=true,
4        parameter="can.Key", condition="!equals")
5     @Group(name="monitoring", selfCompatible=true)
6  })
7  @DefineRules({
8     @Compatible({"join", "monitoring"})
9     @Compatible({"routing", "monitoring"})
10    @Compatible(value={"routing", "join"},
11       condition="!this.isLocal")
12 })
13 public class Peer {
14    private boolean isLocal(Key k) {
15       synchronized (lock) { return myZone.containsKey(k); }
16    }
17    @MemberOf("join")
18    public JoinResponse join(Peer other) { ... }
19    @MemberOf("routing")
20    public void add(Key k, Serializable value) { ... }
21    @MemberOf("routing")
22    public Serializable lookup(Key k) { ... }
23    @MemberOf("monitoring")
24    public void monitor() { ... }
25 }
```

Algorithm 2.3: Definition of groups and rules for a multi-threaded active object. [22]

Language Constructs

The primary configuration mechanism of the system are annotations. Algorithm 2.3 uses the different constructs introduced in [22]. The utilised elements are explained below:

- The `@DefineGroups` and `@Group` annotations are used to establish groups (lines 1 to 6). Thereby, each group can designate a single *group parameter*. The *routing* group (lines 3 to 4) exhibits such a definition with the expression `parameter="can.Key"`.
- Observe the statement `@Group(name="monitoring",selfCompatible=true)` in line 5. The variable `selfCompatible` specifies that *monitoring* requests can be carried out at the same time. Further compatibility rules are defined in lines 7 to 11 through the joint usage of the `@DefineRules` and `@Compatible` annotations.
- The `@MemberOf` annotation assigns a method to the respective group (lines 16 to 23). If a group parameter is present, the argument list of the assigned method has to contain at least one variable of the given type. In case multiple candidates exist, the first one is selected.

Rule Types

A central feature of the model is the flexible design of group compatibility. In this context, the approach supports two kinds of rules:

- *Static Rules* directly specify full compatibility of two particular groups. An example of this type is given in line 8 of the code. `@Compatible("join","monitoring")` indicates that requests of *join* and *monitoring* can run concurrently.
- *Dynamic Rules* limit parallel processing of groups through the use of an *evaluation function*. This method is implicitly invoked at runtime when the decision about the compatibility of two particular requests is pending. The retrieved value indicates whether the joint execution of the corresponding operations is possible.

To facilitate the decision about dynamic compatibility, the group parameter is handed over to the evaluation function. Since the specification of this variable is optional, an evaluation function has either zero, one or two parameters. The concrete amount depends on the involved groups. Hence, if no parameters are received, the decision is based solely on the current state of the active object.

The expression `condition="!equals"` in lines 3 to 4 outlines a dynamic compatibility check. Thus, several *routing* requests are allowed to run at the same time as long as their respective keys are not identical. This condition is implicitly verified by invoking `!key1.equals(key2)`. Another example is illustrated in line 10. The instance method `isLocal(Key k)` is used by the statement `condition="!this.isLocal"` to decide if multiple calls can be performed simultaneously. The method receives merely one argument because only *routing* defines a group parameter.

Criticism

This concept proposes powerful instruments to improve the performance of active objects. However, the chosen mechanisms do also have severe drawbacks. We focus our attention on the following points:

- As can be seen in Algorithm 2.3, the compatibility definitions are quite verbose and confusing. The annotations interfere with the expected layout of the class and significantly lengthen the structure at the top level. This statement concerns the static and dynamic type. In order to mitigate this problem, the group and rule portions could be moved into other places. This modification entails that only the method assignments remain in the class files. Furthermore, the extracted parts could enable global definitions which are available to every active object.
- The names of group parameter types and evaluation functions are delivered as simple strings. This approach prevents the compiler from verifying these elements. For example, a name change of a method would cause runtime exceptions if it is not propagated

correctly. In order to guarantee the existence and suitability of the supplied values, a custom annotation processor is required. It validates the contents of the annotations prior to compilation. This mechanism is supported by Java.

- The use of instance methods as evaluation functions is precarious. They run in parallel to the scheduling thread and can access the components of the active object. Thus, erroneous implementations violate the constraint that only one thread is allowed to control the internal state at any point in time.

- To avoid the previous issue, access to components needs to be synchronised. The implementation of `isLocal` in the lines 13 through 15 satisfies this requirement. However, the envisaged technique imposes concurrency control on the entire active object. This style obstructs fundamental benefits of the underlying concept.

Hence, dynamic rules are especially problematic. They effectively reintroduce the disadvantages of shared-state concurrency into the pattern. As a result, the development becomes more error-prone and complex. Therefore, this type of parallelisation should be avoided.

Active objects that support multi-threading are an interesting concept for enhancing performance without completely reverting back to shared-state concurrency. If executed properly, the model could be extended by a powerful and comprehensible mechanism to parallelise access to the internal state. However, the constructs proposed in [21, 22] introduce new problems. Therefore, further research in this direction will be required before the mechanism is mature enough for widespread use in conjunction with active objects.

2.4 Relevance and Comparison to Related Work

To the best of the author's knowledge, there does not currently exist a well-integrated implementation of active objects for the Java language which provides suitable functionality for their use. The presented topics are interesting adaptations of the fundamental pattern. JCobox's component-based approach is able to simulate active objects. However, it is not designed for combining them with conventional passive objects, and some of the main advantages of the programming model are lost if used solely for this purpose. The concept does also not address the topics and issues explained in Chapter 3. The same holds true for the realisation of multi-threaded active objects.

Apart from the advanced techniques, there are two possibilities to attain comparable behaviour in Java. Firstly, a developer can implement all parts of an active object manually and integrate them into the application to obtain the desired functionality. This alternative introduces a lot of boilerplate code that is tedious to write and maintain [29]. Secondly, any of the existing actor libraries can be utilised to achieve the desired results, for instance Kilim [49]. Such implementations are discussed in detail in [27]. The problem of this approach was already mentioned in Section 1.1. Since message passing is a core principle of the actor model, the concurrent components cannot communicate through method calls. Some other object-oriented languages already have built-in support for active objects. An example is Groovy [9], a language that also runs on the JVM. For the given reasons, a custom solution is required in order to make effective

use of the presented pattern in Java. The goal of this master's thesis is to develop a suitable design and provide the corresponding library implementation.

CHAPTER 3

Result Handling Strategies

This chapter is dedicated to the description of result handling strategies and synchronisation functionality. The active objects explained in Section 2.2 support only one particular interaction pattern. A client asynchronously invokes a proxy method and retrieves a future for accessing the computed value. Although this mechanism marks a good starting point, further invocation types are preferable. The following sections give an overview of various call modes. Different kinds of implementation models are addressed, and the concepts are examined in terms of their suitability for realisation. Additionally, each section outlines a potential use case and provides arguments for and against the feature's integration into the model. Section 3.1 deals with the previously mentioned futures. In Section 3.2, the synchronous call mode is described. It forms the basis of Java object-communication and is therefore considered in this chapter. Section 3.3 discusses completely asynchronous invocations. They are used for sending signals and do not provide a mechanism to obtain the associated result. Section 3.4 explains implicit callbacks. These programmable entities are performed in response to an active execution and have access to the return value. Finally, Section 3.5 introduces a new placeholder type which combines the functionality of futures and implicit callbacks.

3.1 Futures

This technique is fundamental to the pattern and was already explained previously. A method is asynchronously invoked by sending a method request to the corresponding active object. The caller receives a future for acquiring the result and examining the state of the execution. This procedure is outlined in Figure 3.1.

Possible Implementations

The main idea is to associate each method request generated by the proxy with a future. This placeholder is set to the correct value once the scheduler finishes the execution of the servant's operation. Hence, the request class implicitly constructs a future and provides a related getter

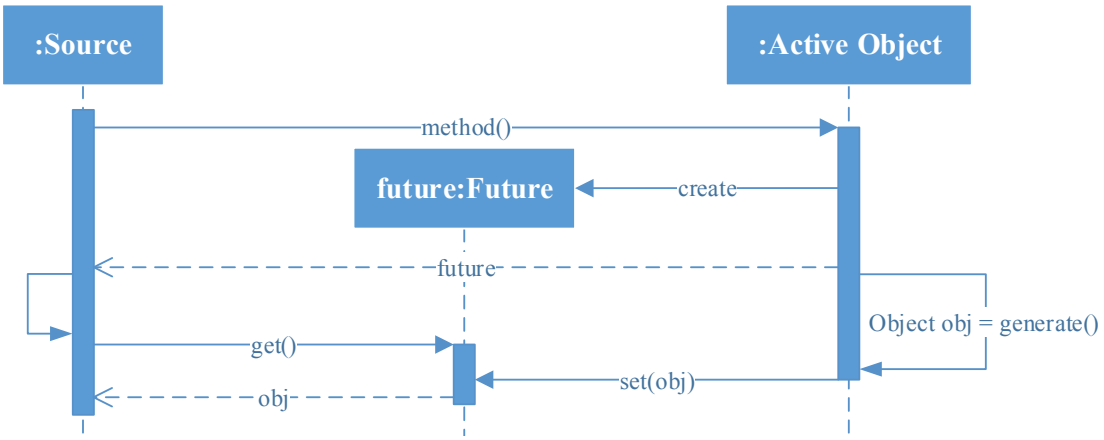

Figure 3.1: Sequence diagram of the asynchronous call mode using a future for result access.

method. The proxy method performs this operation on the newly created object, obtains the future and returns it to the caller.

Considering the basic components, the JDK's `java.util.concurrent` package already offers some interesting types which can be reused. The `Future` interface defines the publicly available functionality of a future. The corresponding `FutureTask` implementation can be utilised either directly or indirectly through inheritance. Another option is to realise an entirely new class based on the interface to gain additional control over the code and diminish the impact of potential change. In Java, futures can also be resolved transparently by using implicit proxies [41]. However, this matter is not taken into consideration, because the developer cannot easily query the execution state of these objects.

Use Case

The startup of an application includes activities that need to be decoupled from the GUI thread. One of them establishes a connection to an update server. This task is easily achieved by using an active object. It maintains the network connection and has an active method responsible for searching for a new version. The thread simply invokes this operation, checks the status of the received future in a regular interval and adapts the GUI as soon as the result is available.

Reasoning

There are several arguments in favour of this mode. The main point is that asynchronous interactions between active objects are an integral part of most applications based on this programming model. The arising future is thereby either temporarily stored or returned to the caller. Its value is usually retrieved at a later date when the execution has already finished. More specifically, this approach has the following merits:

- Futures enable flexible interactions because they avoid implicit synchronisation of the invoked method and the caller. This independence helps to reduce temporal dependency

between distinct active objects, especially in situations where they realise long-running functions.

- Since an active method's invocation is decoupled from its execution, the caller does not necessarily have to wait for the completion of the request. As a consequence, asynchronous invocations lower the response time. They also increase the throughput of the calling method if the result does not have to be resolved immediately.

For the given reasons, this feature forms the base of the library. Although the mechanism is primarily important for the collaboration of active objects, it is still beneficial to programs which utilise the pattern for independent service components.

3.2 Synchronous Calls

The most essential mode with respect to standard Java programming is the synchronous invocation style. When a method of an active object is called, the current thread blocks and waits for the completion of the execution. If a value is returned, it may be assigned directly to a variable. This modality is subject to the basic try-catch-finally semantics.

Possible Implementations

The underlying pattern foresees that the proxy methods return a future. Thus, synchronous calls are realised by retrieving the deferred value in a blocking manner. This task can be completed either directly in the proxy or through a library call. The former option is used to overwrite a servant's operation by a proxy method because the mechanism does not involve any modifications of the related method signature. Hence, this mode is valuable if the proxy is automatically generated by extending the servant.

Use Case

The backend of a hotel's booking application provides a method to make a reservation. An exception is thrown in case the entered request cannot be fulfilled, for instance if the room is already occupied for the given date. To support simultaneous access by multiple terminals, this functionality is encapsulated within an active object. Whenever a client wants to book a room, the servicing receptionist enters the necessary information and submits the reservation. In order to immediately analyse the result and print a notification message, the system invokes the active object's operation in a synchronous fashion. In this example, the asynchronous approach is not necessary, because the user is awaiting the task's outcome.

Reasoning

In typical concurrent Java applications, a large amount of objects is created. Most of these instances do not need any added synchronisation, for example because they are immutable, thread-safe or held solely by one owner. Enforcing the pattern on all types results in an unnecessary

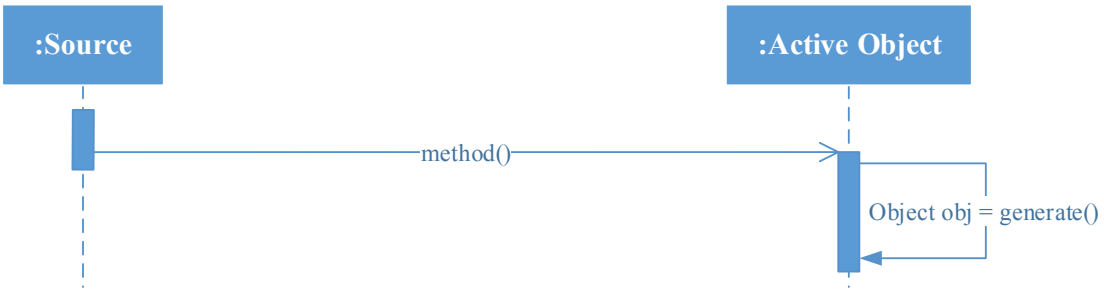

Figure 3.2: Sequence diagram of the purely asynchronous call mode.

amount of threads and impairs the performance of the execution. Therefore, a requirement of the library is the ability to freely combine active and passive classes, even if they are situated in the same class hierarchy. This demand influences the rationale behind the inclusion of the presented feature. In particular, the arguments supporting this mode are as follows:

- The synchronous style facilitates transparency because the default semantics of method calls still hold true from the developer perspective. Furthermore, inheritance obscures the actually used call mode since a normal class can be extended by an active class.
- Class hierarchies using both types become overly complex. If the asynchronous mode is used by default, each method invocation would require a distinction between active and passive objects in the code of the caller.
- Plain objects usually expect standard call semantics to be in effect. In particular, this argument applies to reused components that are not designed for the described mechanisms.
- In the original approach, altering the active property of a class entails a refactoring of all invocations targeting the corresponding instances. If this characteristic propagates downwards in the class hierarchy, such adjustments do also affect any subclasses.

This enumeration is not complete, because it does not consider advanced techniques like class reloading or hot swapping. For the reasons given above, this functionality is implemented at the base level by implicitly resolving the future of a method call.

3.3 Purely Asynchronous Calls

Purely asynchronous method calls send a signal to an active object. Once the transmission has completed, the thread immediately continues with the execution of the following statements. This modality does not offer any way of retrieving the result. The described behaviour resembles the message sending process of the actor model and is similar to the previous asynchronous mode. However, in this case, the future does not need to be generated and handled. It is important to note that when this technique is applied, exceptions raised during the target method's execution cannot be caught by usual means.

Possible Implementations

There are two possibilities for realising this mode:

- Implementation of the proxy methods through the procedure discussed above. When a purely asynchronous call occurs, the future is not returned even though it was already constructed. This alternative is easier to implement because it reuses the existing approach.
- Introduction of a separate mechanism for placing a method request without constructing the associated future. This style implies a lower overhead and does therefore enhance the performance. However, it requires a supplementary method request type and specific handling during the instrumentation phase.

Both options can be used to redefine the actual servant method since it is not necessary to alter the original method signature. Thus, this mode is the second of the two techniques to accomplish automated instrumentation of active objects.

Use Case

Let us remember the booking application mentioned before. It is equipped with a feature to display all current reservations. Assume one active object manages the database connection and provides a method to add a listener for newly created reservations. Using this function, the GUI installs an object as the receiver of these notifications. Whenever new information arrives, the reservation list is updated accordingly. Because this action requires a certain amount of time, access to it has to be controlled. Otherwise, simultaneously processing several notifications can lead to a display error. Therefore, the listener is realised by an active object. To increase the performance of the database object, the listening method is invoked purely asynchronous. This precaution ensures the notifier is not slowed down by a long-running subscriber. Also, there is no interest in the outcome of the active invocation. The only important aspect is that the call is executed eventually, a guarantee which has to be fulfilled by the library in any case.

Reasoning

The potential advantages are not only inherent to the matter but also closely linked to the implementation model. The different aspects are outlined below:

- This style is the sole asynchronous call mode that can be used to overwrite servant methods. It supports sending signals through direct invocations, which is a useful trait for the library implementation. The other mechanisms either return a placeholder object or associate a callback to an invocation. These forms require a modification of the original method signature and are therefore not suited to the task in Java.
- If no future is constructed, the performance and throughput of method calls improves. This enhancement is restricted to situations where the caller is not interested in the outcome of the invocation.

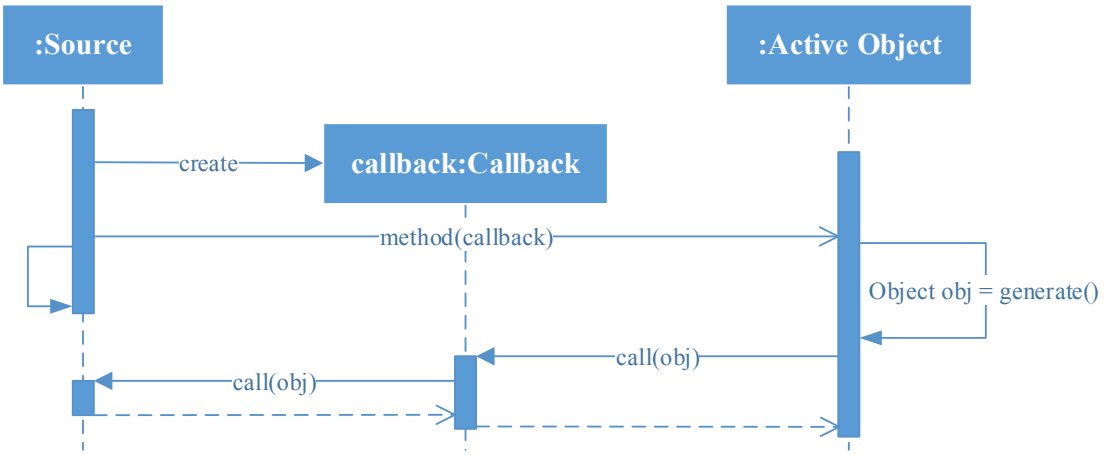

Figure 3.3: Sequence diagram of the asynchronous call mode based on a callback for result handling.

- This approach can be used in conjunction with explicit and implicit callbacks because they do not necessarily require a placeholder object. These concepts are discussed later in this section.

The projected performance increase that comes from omitting the future is slim. Furthermore, only applications with a high number of purely asynchronous invocations are benefited. On the downside, this variant needs a detection mechanism and special treatment inside the model's internal components. Therefore, the implementation of this call mode does not receive particular attention. Instead, it is based on the normal asynchronous procedure presented above, whereby the constructed future is ignored.

3.4 Callbacks

An alternative to the placeholder mechanism is the use of callbacks. These objects are passed as additional parameters to the asynchronous method and are invoked by the callee when the result is available. Although this strategy achieves the desired effect, it requires custom handling of the callbacks and introduces boilerplate code. Furthermore, the target method must directly support this execution type. Otherwise, a separate thread has to be instantiated to accomplish the same task. To solve these issues through a library feature, asynchronous calls have to be linked directly to concrete callback objects. Instead of the manual procedure described above, the system automatically executes the associated callbacks once the target operation returns the result value. Figure 3.3 depicts the sequence of interactions of this approach.

Possible Implementations

This feature can be implemented in a variety of ways:

- Either the method request or the future class is equipped with an instance variable which accumulates the associated callback objects. Once the actual operation terminates, the request's processing method checks this variable. If the collection contains any callbacks, they are scheduled for execution.
- An active object has an additional thread that polls the method requests in regular intervals. As soon as one of them completes, the thread looks up the added callbacks and arranges the execution.
- Every active object receives an additional activation queue for storing the outstanding callbacks together with the respective parameters. To process the incoming jobs, a listener is added to the queue. This entity extracts the callbacks and executes them with the corresponding parameter.

From the above possibilities, the first one is the most sophisticated because it integrates well into the request handling procedure of active objects. However, a correct and efficient realisation involves considerable effort. The other two techniques are easier to implement but entail some overhead. The second approach requires an additional thread for each active object. This circumstance limits the performance of all applications, even when they do not utilise this functionality. The third option has a higher memory requirement since it stores already available information. Furthermore, the listeners have to be treated accordingly. For these reasons, the first suggestion is preferred in the context of the project.

Assignment of callbacks. In any case, the library needs to provide means for installing a callback to an active invocation. These functions have to obtain the underlying method request and perform the necessary tasks for the desired action. For consistency reasons, the assignable callbacks share a common interface. It has a single method representing the reaction behaviour. This operation is performed when the asynchronous execution finishes and receives the return value as a parameter. The class of this argument is specified by the generic type parameter of the interface. The deployment of separate error callback objects is also possible. They handle the erroneous state by acting on exceptions which were thrown during the asynchronous execution.

State Encapsulation. In the context of active objects, the execution of callbacks requires special attention. Imagine the implemented reaction behaviour directly manipulates a variable of the object. Then the callback method cannot be carried out at the same time with other method requests, because parallel execution violates the constraint that only one thread is allowed to access the internal state at any given time. There are two methods for solving this problem:

- The execution of the callback is synchronised with the scheduler's processing method in order to prevent requests from running at the same time.

- The execution of the callback is conducted by the scheduler. In this variant, the task is scheduled like any normal method request object.

The latter option seems cleaner and can be better integrated because it does not introduce additional concurrency constructs into the core classes. It also does not favour callbacks over typical method requests or vice versa. Furthermore, the added synchronisation involves a performance degradation of active objects. Hence, the library's realisation of this feature is based on the second alternative.

Use Case

Imagine a distributed database which needs to integrate a particular record at multiple nodes. If one of them signals an error, the whole transaction has to be reverted on all concerned parties. The system is overseen by a coordinator that manages the state of the transaction. The individual nodes are implemented as active objects. To accomplish the aforementioned goal, the coordinator starts a transaction on all nodes and sends the new dataset to them via an asynchronous call. These invocations are linked to a callback and an error callback. The former counts the incoming confirmations. If all nodes acknowledge the insertion, the callback invokes a commit method on each one of them. However, if any of them fail to include the change, the error callback is executed. Its task is to roll back the transaction on every node.

Reasoning

Although futures are adequate for a multitude of use cases, there are situations where these objects are not appropriate. Particular circumstances can require a reaction to the completion of an active method call. This behaviour can only be realised in two manners when utilising the functionality of futures:

- Invoking a blocking method in order to wait until the result is set.
- Polling the execution state in regular intervals, and requesting the result once it becomes available.

The second alternative suffices if the querying thread can be used in other ways, for example to repeatedly refresh the GUI of an application. However, some active objects only consist of quick methods to achieve fast response times and a high throughput of method requests. In these cases, both of the above possibilities are suboptimal. On the one hand, the first option forces the request processing to pause. This act seriously harms the object's performance. On the other hand, the second alternative requires the future to be stored temporarily. The response has to then be accessed at another place. This approach complicates the structure of methods and is not always feasible.

These issues are tackled by implicit callbacks. The following factors speak in favour of using this type of mechanism in the library:

- An object is able to react to the completion of an active method call without polling or blocking.

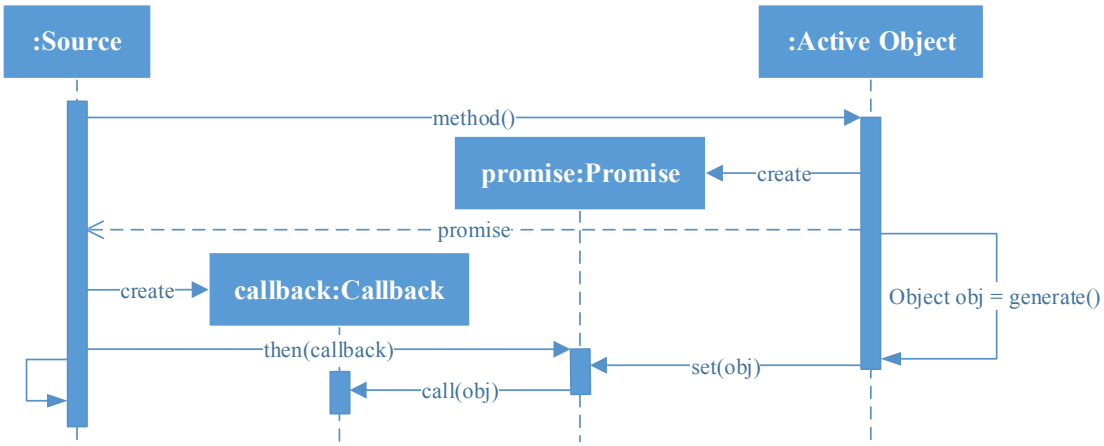

Figure 3.4: Sequence diagram of the asynchronous call mode based on a promise and a callback for result access.

- Any active method can be associated with callbacks, even if it was not originally designed to do so.
- The callback is invoked implicitly by the library, so in most circumstances there is no need to incorporate explicit callbacks into an active object's design.

Limitations. With all the advantages mentioned above, using this technique inside a method has a significant drawback. The difficulty lies in the fact that, by default, the outside clients cannot react to the completion of the callbacks. In order to do so, the argument list of the operation has to include an explicit callback object. The latter features a response method that is manually invoked when the execution of the original callback finishes. As a result, such a parameter has to be integrated into the whole call hierarchy.

Imagine the coordinator of the previous use case is implemented by an active object and the application offers functionality for inserting a new record into the database. In this case, the GUI has to display an appropriate message in response to the completion of the procedure. Because the coordinator uses callbacks for this purpose, it cannot be guaranteed that the new record is committed when the insertion method returns. Therefore, the GUI cannot simply output a notification when the operation terminates. To resolve this shortcoming, the coordinator has to accept an explicit callback object which is executed after the transaction has been committed.

Because of this major problem, the call mode is not implemented directly but is incorporated through a special form of future instead. This type is called *promise*. It is explained in detail in the next section.

3.5 Promises

Promises [24, 26] combine the functionality of futures with the ability to react to the completion of the execution. Similarly, these advanced objects are placeholders for deferred values, and

offer methods for querying the execution's state and result. However, the original concept is extended through a variety of so-called callback assignment methods. They enable connecting different types of callbacks to a promise. As a consequence, the provided callbacks are scheduled automatically once the associated promise is completed. Figure 3.4 outlines the interactions of this procedure.

Promise chaining. An important aspect that adds flexibility and transparency is promise chaining. As previously discussed, some circumstances require reacting to the completion or the result of a callback. Therefore, a new promise is returned whenever a callback assignment method is called:

- This instance enables access to the state of the corresponding execution and provides functions for requesting the result, if any is returned.
- The new instance also has exactly the same callback assignment methods. Calling one of them generates another promise.

This mechanism supports chaining together multiple callbacks to realise a variety of situations. For example, it can be used to process an asynchronous invocation step by step. Instead of delivering callbacks explicitly as a parameter, the developer may decide to pass back a promise. A caller can then invoke one of the callback assignment methods and return the resulting promise. The instance enables other components to await the result or install implicit callbacks. This practice is transparent because the receiver is not aware of the promise's predecessors or of the length of the chain.

Possible Implementations

To realise the proposed concept inside an active object library, the internally used futures are entirely replaced by promises. The new technique is an expansion of the original idea and, most importantly, does not remove existing functionality. Hence, the substitution does not affect any established applications. Apart from the implementation of the added classes, the core concepts remain essentially unchanged. Only a few adaptations of the model are necessary.

A central aspect in this context is the appropriate result treatment. As soon as a promise is completed, the system has to look up all of the associated callbacks and schedule their execution. Because a simple Boolean expression is sufficient to check for the existence of children, little overhead is involved when promise chaining is not used. Preferably, the descendants are handled implicitly by the promise when its execution result is set. As mentioned above, a scheduler has to be able to conduct the execution of the associated callbacks. This mechanism is necessary when the internal components of an active object are directly accessed by the reaction behaviour.

Use Case

A transformer component is responsible for the retrieval and preparation of data records on behalf of the representation layer. The database access is controlled by an active object, which features a method for obtaining data sets through the execution of a query. The transformer

calls this function asynchronously with an appropriate query and retrieves a promise for the result set. In the next step, a callback is set up. It performs the modifications and passes back the manipulated objects. The transformation method then returns the promise obtained from invoking the callback assignment method. The GUI receives the procedure's result either via blocked waiting or by reacting to the completion of the promise inside a callback. The latter task can be achieved through an anonymous inner class that updates the GUI components directly.

Reasoning

Since promises are a mix of futures and callbacks, the approach inherits their combined advantages. Furthermore, the concept adds the following benefits:

- Through promise chaining, complex tasks can be decomposed into several asynchronously executed steps. This strategy is especially useful if the whole process is not implemented by a single component.
- In contrast to the simple implicit callback approach, this mechanism creates a new promise instance when a callback is installed. This object can be used to check the status of the execution, retrieve the result and add further computation steps.
- As already mentioned, promise chaining is transparent. The predecessors of a particular promise object are concealed in order to give simple and concatenated promises a uniform appearance.

Aggregated promises. Although promise chaining is useful in a variety of situations, it does not solve our problems entirely. Recall the program that uses a distributed database for storing data. The various nodes are implemented by an active type. It provides a method for preparing the integration of a new record into the database. An exception is thrown if the data set cannot be incorporated. The coordinator is also represented by an active object. It has a method for inserting a new record. This functionality is accomplished by asynchronously invoking the insertion method of the individual nodes and relating callbacks to each promise. Because the coordinator retrieves multiple promises, there is no single instance which can be returned to the client. These kinds of situations require manual effort to create a custom object. It is updated by the callbacks on success or failure. Alternatively, the library has to offer multiple generic aggregation methods. These functions are not considered in this thesis.

Nested promise types. Another related problem are nested data structures. It applies to futures as well as promises. As an example, observe the promise type that wraps a promise to an integer (cf. the java type `Promise<Promise<Integer>>`). The issue occurs when an active method which returns a promise is called asynchronously. As a consequence, a new promise repacks the original one. This phenomenon can entail deeply nested data structures. They are not only hard to handle, but do also complicate the maintenance of software components. A library has to provide means to counteract this problem.

CHAPTER 4

Modelling of Selected Functionality

This chapter concerns the design of the core components outlined in Chapter 2 and presents the realisation of the functionality described in Chapter 3. Section 4.1 explains the fundamental implementation principles that constrain the model and gives reasoning about their relevance. In Section 4.2, the components' functional requirements, relationships and behaviour are covered. The descriptions are augmented by diagrams to picture the relevant situations. Towards the end of this part, the presented model is contrasted to the cited literature in Section 4.3. Any deviations from the base pattern are highlighted and arguments for the decisions taken in the design process are given. In general, the differences are necessary because the application is transparently enriched through bytecode instrumentation during load-time. Hence, the written code is subject to the rules imposed by the Java compiler.

4.1 Fundamental Ideas

Before demonstrating the created model, we address the basic ideas behind the library. They restrict the layout and the relationships of the components. These principles serve to justify the made decisions and are used during the discussion at the end of this chapter.

Automation and Maintainability

The most significant aspect of the resulting library is the accomplished level of automation. The base pattern requires a large quantity of boilerplate code in the form of proxy classes and method request subtypes. Each of the active methods in the servant class needs a corresponding proxy operation that handles occurring invocations and dispenses new futures to its callers. Another notable feature are *passive methods*. They directly invoke the appropriate servant method instead of taking the detour through the scheduler. This mode is suitable for retrieving immutable, final objects[1] because direct access to them does not violate any synchronisation constraints.

[1] Immutable objects cannot change their state after creation. Since their associated reference is final, it may not be adjusted after it is initially set.

Generally speaking, methods which do not enter a critical section can be kept passive. The following segment illustrates the structure of a proxy class in the original pattern:

```
public class CounterProxy {
  private Counter servant; //The object doing the actual work
  private Scheduler scheduler; //The scheduler of
                               //the active object
  {...}
  //Demonstration of an active method inside a proxy:
  public Promise<Integer> add(Integer increment) {
    MethodRequest request = new AddMethodRequest(servant,
          increment); //Create request for invocation
    scheduler.insertRequest(request); //Insert request into
                                      //activation queue of scheduler
    return request.getPromise(); //Return associated promise
  }
  //Demonstration of a "passive" method inside a proxy:
  public int hashCode { return servant.hashCode(); }
}
```

Some details were omitted in the given sample, but it visualises the central problems of the chosen approach.

- Each proxy method is composed of a basic set of code fragments, as can be seen in the displayed `add` operation. These parts are only minimally adapted depending on the situation. For example, the request subclass is typically exchanged.
- A large number of active methods makes the creation of an equal amount of method request subtypes necessary. We will spare the concrete details of these classes. However, they introduce additional boilerplate code, which is not to be underestimated.
- Passive methods must delegate calls to the associated servant instance because the proxy does not inherit the servant's operations. For example, observe the `hashCode` function in the presented extract. This obligation limits the interchangeability of the approach. Replacing a servant by its proxy or vice versa is a serious effort because the substitution invalidates all existing references and method calls.

For the given reasons, the base pattern is not simply adopted. Instead, the proposed model adjusts the structure of the components and the interactions in order to provide a solution better suited to our purposes. In this context, two goals have to be highlighted.

Boilerplate code. A high priority is the reduction of *boilerplate code* [29, 30]. The optimal solution entails the complete elimination of unnecessary code. This notion enables the programmer to focus his full attention on implementing the actual functionality. To this end, the configuration of active objects is performed through the use of annotations and potentially empty interfaces.

Interchangeability. Another vital characteristic is the aforementioned *interchangeability*. Ideally, a worker class can become an active class without invalidating any existing code. On the contrary, a developer should be able to turn active objects into passive instances without causing great amounts of compilation errors.

Utilisation of Tools

Another limiting factor is the manner in which the desired results are achieved. There are several possible tools to this end. Java offers an integrated approach to metaprogramming through the Reflection API. Moreover, various third-party libraries exist. They can be used as precompilers or offer the ability to perform bytecode manipulations. We now take a brief look at the benefits and drawbacks of the available options, and give reasoning behind the made decisions.

Code Generation. The central task is the creation of proxy classes and other code sections needed to attain active objects. Although the JDK facilitates the analysis of the underlying types, the ability to produce or modify classes through the standard API is limited. For the purposes of code generation, the use of precompilation, bytecode manipulation or a combination thereof is advised. From a technical standpoint both approaches alone suffice to satisfy the needs of the library. But it should be noted that there is a lack of sophisticated precompilitation instruments in the Java world. During an investigation, only one framework was discovered which was considered adequate to reach the objective. It is named Polyglot [4, 40] and it was also used to implement JCobox [45]. However, in its current form, the tool does only support Java 5. In contrast, a number of suitable bytecode instrumentation libraries exist. Popular examples are ASM [8], BCEL [14] and Javassist [11, 12]. They are frequently upgraded and support more recent Java versions and features. Therefore, bytecode manipulation is chosen as the method of code generation. We have selected Javassist despite its limitations [50], because it offers a high-level interface to perform transformations. Furthermore, the referred restrictions carry no weight in our use case.

Introspection. As previously mentioned, the JDK provides its own share of metaprogramming facilities. A disadvantage of reflective programming is the added performance overhead. On the positive side, the JDK directly supports these mechanisms. Furthermore, they enable the development of scenarios that cannot be achieved by other means. Since we want to make use of annotations to tag active objects and active methods, the utilisation of the integrated introspection features is a must. In several situations, one of the internal components needs to analyse a particular property of another type. Simply put, there is no real alternative to the reflective capabilities provided by Java. Thus, reflection plays an essential part in the functioning principle of the library.

Ease of Use

Although usability is an important characteristic of programming libraries, research is difficult in this area due to the user-centered subject. Clarke [13] discussed criteria and questions taken into

account in the development process of this project. Because the implemented API has specific needs in this regard, we address the applied guidelines next.

Unintuitive design. A crucial point is that no facilities of the API offend the comprehension of an experienced developer, especially because bytecode instrumentation makes unorthodox designs feasible. Such implementations impede maintenance since they result in incomprehensible and complex source code. The sample below illustrates a peculiar realisation:

```
public class ActiveCounter {
   public Integer add(Integer increment) { ... }
}

ActiveCounter counter = ...;
Promise<Integer> promise = Active.callAsynchronously(counter.add(5));
```

This code assumes the `ActiveCounter` class has an active method named `add`. A programmer expects the statement `counter.add(5)` to be executed prior to the invocation of `callAsynchronously`. This ordering makes the latter call ineffective. However, bytecode manipulation can give this simple program extract an entirely different meaning. Imagine an instrumenting component searches the bytecode for occurrences of the aforementioned static method `callAsynchronously`. If the invocation is encountered while parsing this class, the enclosed expression is replaced by an appropriate active call. In this case, the `add` method is substituted by the corresponding proxy operation. This mechanism is not only confusing but also raises serious doubt as to the library's capabilities. Therefore, inconsistent designs have to be avoided.

Straightforwardness. An additional factor related to usability is the simplicity of the API. If a certain feature is complicated or inconvenient to use, a programmer will refrain from using it and instead develop workarounds. Hence, the individual library components have to be meaningful and relatively easy to use. Once again, the optimal fashion of creating an active object is with the help of annotations and possibly empty interfaces. These techniques do not automatically involve changes in existing code or cause further problems. Setting up active methods can also be done through annotations. Functionality that is not solvable by other means is combined into a handful of utility classes for easy access.

Documentation. An often overlooked but largely important facet of library usability is the documentation of the components [42, 43]. For this purpose, the core classes are described with the aid of the software documentation tool *JavaDoc*. It is directly integrated into the JDK and can generate HTML files from source code comments. Cross references are used to clarify the relations between various types. Another important part of the documentation are example implementations. They illustrate the use of the API and outline the interactions of the different classes. Chapter 6 provides such sample code and discusses the advantages and disadvantages of the chosen approach.

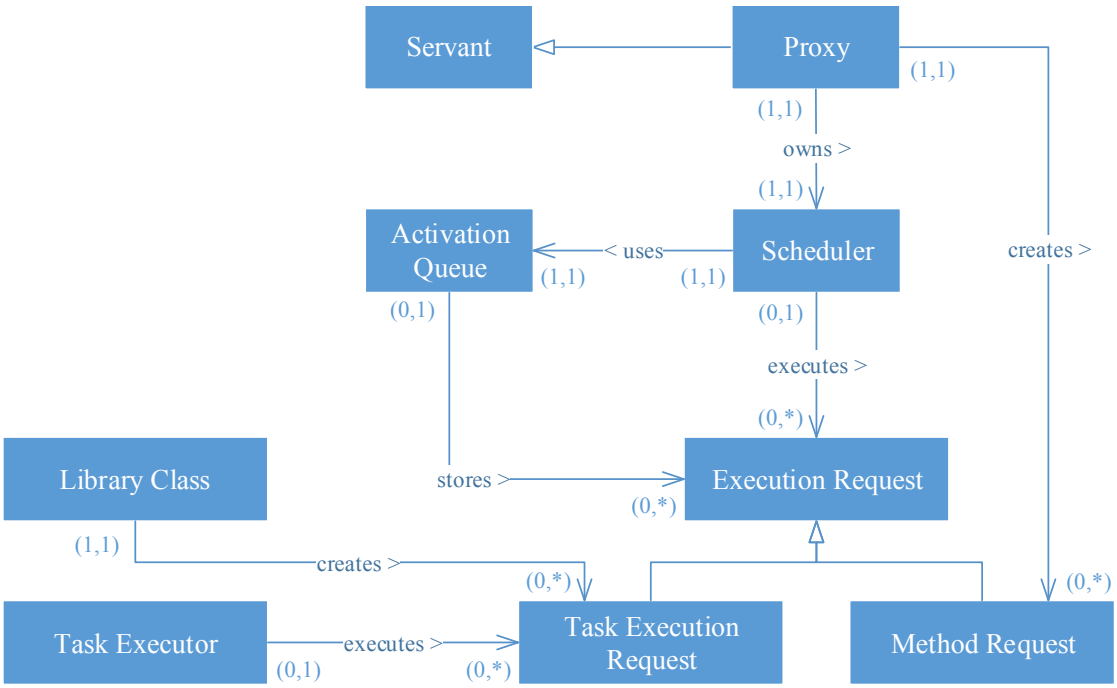

Figure 4.1: Class diagram depicting the relationships between the different core components.

4.2 Core Components

The component modelling is guided by the literature showcased in Chapter 2. Since the library is intended to support promises, they entirely replace the previously noted futures. Figure 4.1 exhibits the individual elements and their connections. A large part of the properties stem from the active object pattern, while some characteristics have been refined. The rest of this section is dedicated to the explanation of the components and the reasoning of deviations from the basic pattern. The JavaDoc of the respective class [2] gives more information about the operational behaviour. It is noteworthy that the description in this section is not a thorough specification of the components' structure. In particular, the concrete implementation is not compulsory. The developed code presented in Chapter 5 embodies a proof of concept realisation of the described model.

Promise

As already explained in previous chapters, a promise is a container object for the result of a deferred execution. The corresponding `Promise` interface extends the `Future` interface located in the `java.util.concurrent` package. This relationship increases the compatibility with the JDK and the basic active object pattern. Figure 4.2 outlines the declared methods of these types and indicates the inherited operations.

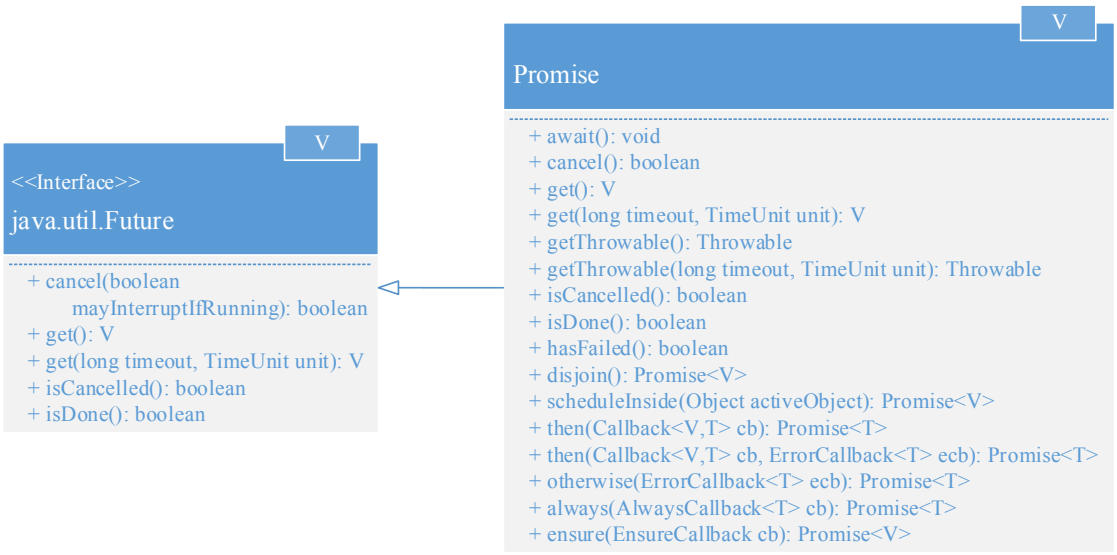

Figure 4.2: Class diagram of the modelled promise interface.

Promises are primarily used to enable access to the outcome of active methods, callables and callbacks. In this context, the former two elements represent the starting point of a computation, whereas the different sorts of callbacks are registered by calling the appropriate methods of an existing promise. The various kinds of features provided by this interface are stated in the following subsections.

Completion

The library requires suitable functions in order to complete a promise. They are not present in Figure 4.2 because they are mostly used internally. These operations comprise setting the result of an execution and can also entail further instructions. Although this model does not dictate the manifestations of these methods, it defines the effects of the completion of a promise. They are listed next:

- The state of the promise is adjusted accordingly.
- Any thread performing a blocking method is unblocked and notified of the outcome of the operation.
- Calls to result retrieval methods obtain the computed value immediately without blocking.
- All associated callbacks are scheduled automatically.

Thereby, an adequate treatment of the installed callbacks is necessary. If they directly access or modify the state of an active object, their execution must be conducted by the associated scheduler to avoid race conditions. At the same time, every non-intrusive callback has to be invoked

by a separate thread to increase the throughput of the active objects. The responsibility for the selection of the correct type is assigned to the developer through adequate library functions.

The following enumeration summarises the behaviour of the state retrieval methods upon completion of the related promise:

- `isDone()` returns `true`.
- The value of `hasFailed()` depends on the manner in which the computation finished. It evaluates to `true` if the invocation raised an exception or was cancelled. If it terminated normally, `false` is passed back.
- `isCancelled()` yields `false` in case the promise was correctly completed and not cancelled beforehand.
- Subsequent calls of the state retrieval methods of a finished promise always have to return the same value.

Cancellation

The `cancel` operation is used to abort the task associated to the promise object. This feature is adopted from the `Future` interface. The method receives a single Boolean parameter and returns a Boolean. The call semantics are explained in the list below:

- `cancel`'s result signifies if the promise was successfully cancelled. This function is only effective in the initial or running state. Then `cancel` returns `true`. Already completed or cancelled tasks cannot be aborted. Hence, `false` is returned in such cases.
- The parameter `mayInterruptIfRunning` indicates if `cancel` interrupts a running task. In case the parameter is set to `true`, an interrupt signal is sent to the thread that processes the corresponding invocation. This intervention may be necessary when the runner is expected to perform a blocking command. If `false` is passed to the method, the executing thread is allowed to terminate normally without being disrupted by a signal.
- The method does not throw any unchecked or checked exceptions.
- The cancellation of a promise causes all of its child promises to be cancelled as well. The reasoning behind this decision is that an abortion represents exceptional behaviour. In this case, there is no value that may be passed unto the children. Even though the establishment of a cancellation callback is a noteworthy option for corner cases, this mechanism is not being considered in the model.

Result Retrieval

Naturally, a promise offers methods to access the computed result. These operations are described below:

- `get` retrieves the return value, if present. A checked `ExecutionException` object is raised if the associated execution failed due to an error. The real exception instance is stored as the cause of the `ExecutionException` and may be retrieved by invoking `getCause()`. In case the promise has been cancelled, the operation throws an unchecked `CancellationException`. This behaviour is guided by the specification of the `Future` interface.
- `getThrowable` can be used to request the exception that was raised during the execution. The function either yields the actual instance or, if none occurred, `null`. An object of `CancellationException` is passed back for cancelled promises.

Both methods exist in a variant that halts indefinitely and a version enabling bounded blocking. The latter requires the specification of a timeout. When this period is exceeded, a checked `TimeoutException` is thrown. The exception is utilised to distinguish successful from expired invocations.

Execution State

As already mentioned, the `Promise` interface contains methods for requesting the status of the execution. These operations return a Boolean value which indicates the corresponding state. They are now explained in greater detail.

- `isCancelled` queries the cancellation state. This method yields `true` if the execution of the related task was successfully cancelled. Otherwise, `false` is returned.
- `hasFailed` gives information about the result of the execution. The function evaluates to `true` when the associated job has failed. This outcome means the call was cancelled, or it terminated due to an exception. `hasFailed` resolves to `false` if the execution completed normally or has not finished yet.
- `isDone` signals the completion state of the promise. `true` is passed back if the underlying invocation has finished. This value does also concern failed or cancelled tasks. The result is `false` in case the execution has not started yet or is still running.

Callback Assignments

A noteworthy enhancement compared to futures are the callback assignment methods. They can be used to set up specific kinds of callbacks in order to react to the finalisation of a procedure. Through the support of promise chaining, composed computations are within the realms of possibility. This mechanism also provides access to the outcome of the callback's execution.

- The variations of `then` and `otherwise` install several combinations of `Callback` and `ErrorCallback` instances to a promise. The former react to successful completions, while the latter handle failed terminations. The different operations make the specification of both or only one of the types possible.
- An instance of `AlwaysCallback` is installed by calling `always`. This sort handles normal and erroneous completions. Because the callback method has to address both result and exception objects, it must accept a parameter of the plain `Object` type.
- The `ensure` operation accepts an instance of `EnsureCallback`. This form is invoked regardless of the previous promise's outcome. The corresponding callback method does not receive any parameters and it does not return a result. Similarly to Java's `finally` construct, this mechanism can be used to close open files or network connections.

In the context of processing the next level of the promise chain, the model relies on two important techniques:

- The final value of a promise is automatically assigned to descendants which do not possess an appropriate callback. For example, this situation applies to a child that only has an error callback, even though the previous task finished normally.
- The model respects the execution context of callbacks. As already noted, some of them must be conducted by a specific scheduler. This approach is necessary to guarantee state encapsulation in particular circumstances. On the contrary, other callbacks do not need controlled treatment. Hence, they have to be processed by a unit of an independent thread pool. As an example, this modality is appropriate for logging the termination of an invocation.

ExecutionRequest

An instance of `ExecutionRequest` encapsulates the execution behaviour of a particular kind of action. This interface is an intermediate layer between the different concrete request classes and the schedulers. It consists of a single method named `execute` which neither has parameters nor a return type. The implementation requirements are outlined next.

- The `execute` operation does not throw any checked exceptions. It is also expected not to raise unchecked exceptions upon activation. The handling of such errors in the invoking thread is not intended and represents undefined behaviour.
- A realisation must intercept all possible exceptions and has to react to these unusual states accordingly, without the knowledge of the scheduler. Thus, an implementing class does not only contain the logic to carry out an action but also to treat undesirable scenarios.
- Successful and erroneous outcomes have to be handled automatically. This treatment includes the invocation of the callbacks associated to the underlying promise. Furthermore, the result must be propagated to child promises which do not provide a suitable callback.

The central implementations of `ExecutionRequest` are `MethodRequest` and the various subtypes of `TaskExecutionRequest`. These variants are described in the following paragraphs.

MethodRequest

A `MethodRequest` object instructs the execution of an active method. Such instances are implicitly created by the corresponding proxy operation. The `MethodRequest` class is thereby used universally to represent all sorts of active methods. As a result, the individual operations do not have their own subtypes. The semantics of performing a method request are summarised in the listing hereafter:

- Calling `execute` triggers the appropriate servant method. Thus, `MethodRequest` must store the passed parameters and supply them to the invocation.
- Each method request uses an associated promise for storing the outcome of the servant method execution.
- As noted in the last paragraph, any installed callbacks require appropriate handling.

TaskExecutionRequest

`TaskExecutionRequest` is another subtype of `ExecutionRequest`. This kind of object orders the execution of `Callable` and `Callback` instances. Because these classes need different handling, they receive individual derived request types. `TaskExecutionRequest` is kept as an abstract class and covers common features of the specialisations. The characteristics of these components are mentioned next.

- A task execution request regarding a callable is created through library functions. These jobs can be initiated immediately because they do not expect any parameters.
- A task execution request concerning a callback is constructed by the callback assignment methods. These instructions can only be performed after the preceding promise has completed since they operate on its value.
- Both forms have a related promise for saving the execution outcome.
- Again, attention has to be paid to installed callbacks and the associated promise children.

ActivationQueue

The `ActivationQueue` type acts as a buffer for incoming execution requests. Its operations and relationships are shown in Figure 4.3. Each queue is directly owned by a single scheduler. The proposed model does not provide a mechanism to share an instance between various active objects. The interface offers the following functionality:

- `insertRequest` adds the supplied execution request to the internal storage.

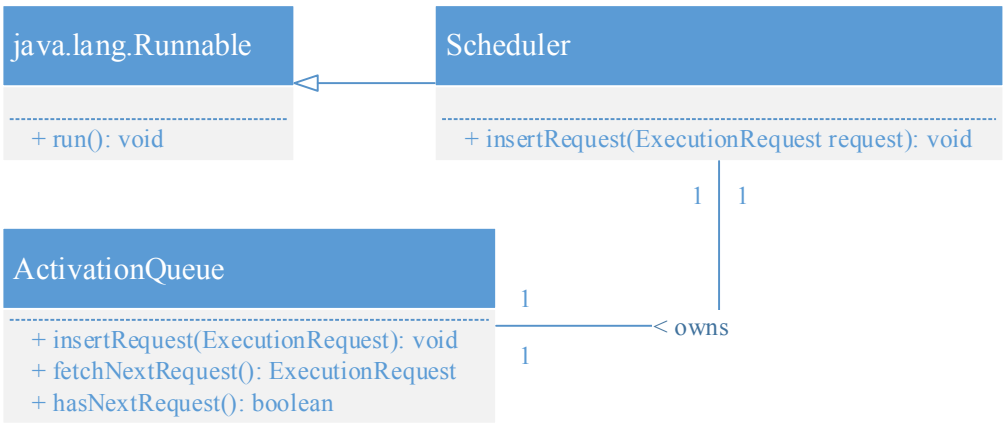

Figure 4.3: Class diagram of the scheduler and activation queue interfaces.

- `fetchNextRequest` retrieves the next request and removes it from the object store.
- `hasNextRequest` indicates if the queue holds any remaining instances. The method returns `true` in case there is at least one request left, and `false` if the collection is empty.

The model does not dictate an ordering of the contained elements. Hence, alternative implementations can dequeue requests in an entirely different order, even if the objects have been enqueued in the exact same sequence. For example, a class can pursue a FIFO (first-in, first-out) or a priority-driven strategy. Depending on the particular realisation of `Scheduler`, classes based on this interface may need to be thread-safe. See Chapter 5 or the provided JavaDoc [2] about these details.

Scheduler

A `Scheduler` instance has to process the received execution requests in its own private thread. To that end, the interface expands the JDK's `Runnable` type. The methods and relationships of this component are depicted in Figure 4.3. The list given below summarises the characteristics that apply to the associated implementations.

- Each scheduler does exclusively own a single `ActivationQueue` object for temporary storage of the execution requests. This collection must not be shared and has to be protected from direct external manipulation.
- The provided `insertRequest` operation enqueues the attached request into the activation queue. There is no support for the external retrieval of requests.
- The inherited `run` procedure is responsible for executing the incoming requests. Therefore, it dequeues the individual objects from the held queue and invokes them.

In addition to these points, the JavaDoc of any concrete realisation of `Scheduler` has to contain a statement about thread safety requirements of the utilised `ActivationQueue` classes. This clarification is necessary because `insertRequest` and `run` access the internal instance simultaneously.

Termination of the scheduler's thread. A crucial aspect is the termination of the active objects. In Java, an application cannot shutdown normally while a thread is still active. Therefore, the `run` method of every scheduler has to end eventually in order to enable a controlled completion. On the other side, each active object needs to be in permanent stand-by. In particular, the insertion of a request must automatically spawn a new processing thread if necessary. As an example, imagine a race condition which prevents a new thread from being started in certain situations. If the active object receives a vital invocation regarding the release of a resource and these specific circumstances are met, the entire application is halted due to a deadlock.

TaskExecutor

As mentioned previously, some instructions do not have to be conducted by a particular scheduler. A `TaskExecutor` is an entity for handling these kinds of tasks inside a separate thread. This approach is appropriate for a multitude of circumstances, most notably logging of raised exceptions or transforming the output on behalf of a client. Similar to the `Scheduler` type, this component is based on the `Runnable` interface provided by the JDK. The implemented `run` operation has to traverse every branch of the promise chain and analyse the discovered elements one after the other. There are three possibilities to treat an encountered object:

- An executable task is immediately carried out. In this case, no further action is necessary because the associated promise is implicitly set to the correct value.
- If the task cannot be performed for some reason, the corresponding promise is set to the result of its predecessor (cf. propagation). As an example, this situation applies to jobs which do not feature a suitable callback type.
- If the task has to be conducted by an active object, the executor sends an appropriate execution request to the concerned scheduler. As a consequence, `run` does not pursue this part of the chain and continues with another branch.

Proxy

This component decouples the invocation of an active method from the corresponding execution. For this purpose, the proxy extends the actual servant class and overwrites the individual active methods. The main reason for this approach is interchangeability, so instances of the proxy class can be used in places that accept a servant object. This relationship minimises the number of necessary changes and avoids compatibility issues. Otherwise, the bytecode instrumentation would break the code and cause other unpredictable problems. Another benefit is that calls of passive methods do not have to be forwarded to the original object. The ensuing enumeration summarises the attributes of this entity.

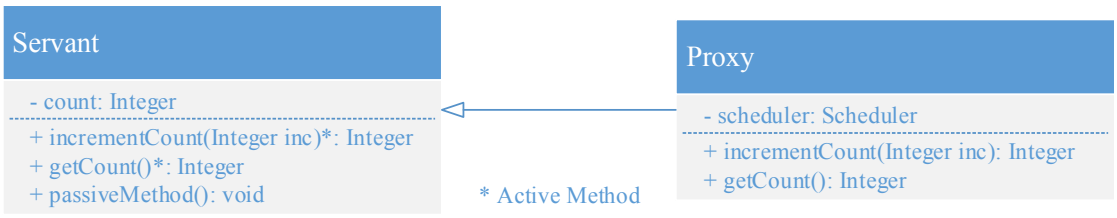

Figure 4.4: An example of a concrete proxy.

- All proxy objects possess a `Scheduler` instance for processing the emerging execution requests.
- To reach the optimal degree of automatism, the proxy type is generated on the fly and every creation statement concerning the servant's class is replaced with an instantiation of the newly created proxy. Thus, the proxy must exhibit a matching constructor for each one available in the servant.
- All of the proxy's constructors are structured in a similar manner. First they invoke the appropriate `super` constructor and then they initialise the internal components. The set-up includes the generation of the `Scheduler` instance.
- Methods marked as active are overwritten by an analogous proxy method. It takes care of the lookup of the associated servant method, creation of the method request and insertion of the request into the scheduler's activation queue.

The proxy's layout for a counter class is exemplified in Figure 4.4. The servant contains two active methods and one passive method. As a result, the proxy contains two operations. They correspond to the active methods of the servant.

The chosen proxy realisation imposes two notable limitations on the developed classes:

Constructor safety. As previously mentioned, the proxy's variables are initialised after the servant's constructor has completed. In this context, active methods can only be called once the associated object is fully initialised. Therefore, the constructors must not perform a call that triggers an invocation of any of the instance's active methods. Otherwise, the execution of the parent constructor causes runtime failures and unexpected behaviour.

Synchronous methods. Active methods with a return type are a challenging matter. Since they need to be overwritten, the corresponding proxy operation has to wait for the result of the invocation. Otherwise, the outcome is inaccessible. This task is fulfilled by extracting the related promise and retrieving the return value through the blocking `get` operation. Hence, these overwritten methods can only be directly used for synchronous invocations. Asynchronous calls have to be provided through library functions. For this reason, purely asynchronous implementations are restricted to procedures with no return type (cf. `void`).

4.3 Deviations from the Active Object Pattern

We now take a closer look at the deviations from the literature presented in Chapter 2.

Proxy method signature. The first and most obvious adaptation concerns the signature of the proxy methods. In the original pattern, they return futures. As a consequence, the presented model has to use promises because it supports them at the base level. Sadly, this approach is not possible. Recall that the proxy is derived from the servant. Therefore, every active method needs to be overwritten by a proxy operation in order to prevent direct access to the servant's definition. In this case, changing the return type to the unrelated `Promise` interface violates the substitution principle of subtype polymorphism. Sections 8.4.5 and 8.4.8.3 of the Java Language Specification [19] explain the notion of *return-type-substitutability* in great detail. This difficulty can only be avoided through a precompiler. However, the use of this technique introduces further problems. They were already mentioned in Section 4.1 during the discussion of code generation.

Removal of `guard`. The second difference in comparison to the base pattern is the absence of the `guard` method in the `MethodRequest` type. Originally, this function is used by the scheduler to determine if certain constraints are satisfied before executing a request. For this purpose, `MethodRequest` is extended for each active method requiring special handling. The subtype simply redefines the `guard` to achieve the desired behaviour. However, in the developed design, `MethodRequest` universally represents all active method requests. A user of the library cannot control the creation of these objects by simple means since the model does not contain a mechanism for the use of custom `MethodRequest` classes. Because a `guard` method cannot be overwritten, it is ineffective and has therefore been removed.

Introduction of `ExecutionRequest`. The use of the `ExecutionRequest` interface is another distinctive feature. In the proposed model, the scheduler processes instances of this type. On the other side, the original pattern directly uses and invokes method requests. This simplification is not possible in the design at hand because it supports callables and callbacks. Since they may need to be executed within the bounds of an active object to guarantee state safety, the scheduler has to deal with multiple kinds of requests. Hence, the `ExecutionRequest` interface was established as an intermediate layer. It provides a unified entity that is processable by `Scheduler` instances.

CHAPTER 5

Implementation of the Model

This chapter presents an active object library implementation for Java. The developed solution is a proof of concept translation of the model proposed in Chapter 4. This realisation is not optimised to exploit its full performance potential and does not meet any of the optional functional requirements mentioned earlier. In Section 5.1, the library's technological preconditions are discussed. Section 5.2 describes the implementation of the components shown in Section 4.2. Thereafter, the instrumentation process and the involved entities are treated in Section 5.3. Section 5.4 is dedicated to the workflow executed by the classes undertaking the transformation. Lastly, a selection of the conducted bytecode manipulations is detailed in Section 5.5.

5.1 Prerequisites

This section addresses the requirements and settings of the presented implementation. Although compatibility with other versions is likely, the library has only been tried and tested using the following tool set:

- Java Development Kit 1.8.0 Update 0
- Java Runtime Environment 1.8.0 Update 0
- JUnit 4.11
- Hamcrest Core 1.3
- Javassist 3.18.0 GA
- JUnitParams 1.0.2

In order to make use of the provided services in an application, it has to incorporate the Java archives (JAR) of the library and of Javassist in the classpath on startup. Furthermore, the library JAR has to be registered as a Java agent to be able to perform the instrumentation. This configuration is illustrated in Section 5.3.

```
1  @ActiveObject
2  public class Container {
3      {...}
4      @ActiveMethod(Synchronicity.Message)
5      public void add(Object object) { ... } /* Adds object to container  */
6      @ActiveMethod(Synchronicity.SynchronousCall)
7      public Object retrieve() { ... } /* Fetches next object from
           container and removes it                                          */
8      @ActiveMethod(Synchronicity.SynchronousCall)
9      public int size() { ... } /* Returns the number of objects currently
           in the container                                                  */
10
11 }
```

Algorithm 5.1: Exemplary usage of annotations in an active container class.

5.2 Implemented Components

The next segment deals with the implementation of the core components described in Section 4.2. Due to the library's dimensions, we only focus on interesting pieces of the realisation here. Code samples and extensive explanations are provided where considered appropriate. The whole code base is available at the referenced repository location [3].

Annotations

A vital facet are the supported annotations. They represent a straightforward technique of denoting classes and methods. During load-time, the agent searches for these markings to locate the parts requiring transformation. The discovered elements are then processed accordingly. This subsection discusses the various annotations in detail. Their usage is outlined in Algorithm 5.1.

`ActiveObject`

The `ActiveObject` annotation serves to designate active object types. It is one of the main mechanisms to interact with the system. The notable aspects of this component are summarised below:

- `ActiveObject` contains no member variable types and can be applied only to classes and not to single constructor calls. The transformation of individual instances into active objects is a reasonable enhancement regarding future developments.
- The annotation is retained even during runtime in order to enable the instrumentation of newly loaded classes. Otherwise, the agent cannot always decide if they create active objects.
- Usually, a type labelled with `ActiveObject` contains one or more methods annotated by `ActiveMethod`. This circumstance is, however, not essential.

`ActiveMethod`

Unsurprisingly, the `ActiveMethod` annotation is used to denote active methods. Only active classes (cf. `ActiveObject`) are examined for these kinds of operations. `ActiveMethod` features an element of the `Synchronicity` enum to determine the behaviour of direct invocations. This type defines two values:

- `SynchronousCall` indicates synchronous invocations. They block the corresponding thread and wait for the completion of the active execution. This mode is used by default when no value is specified.
- `Message` signifies purely asynchronous calls. They return immediately after inserting a method request into the scheduler's queue, and no promise is retrieved for result handling. Exceptions cannot be propagated to the caller.

In the library, the `Message` mode is limited to operations without a result type (cf. `void`). As already mentioned in Chapter 3, the promise is still being created internally, although it is not passed back to the caller. Another interesting characteristic is the definition of passive methods. Operations without the `ActiveMethod` annotation are executed directly even from the outside. This technique allows a developer to integrate concurrent processing of certain procedures into an active class.

Intra-object communication. In this context, a proper handling of object-internal active invocations is necessary. They target an active method of the currently processing instance, and are identified by checking if the executing thread matches the one of the destination scheduler. Such calls have to be performed directly because their scheduled, synchronous execution leads to a deadlock.

`protected` methods. As yet, `ActiveMethod` is only effective when used on operations declared as `public`. `protected` active methods are an idea worth considering in the future because they can be incorporated in recursive data types. For example, this mechanism is beneficial to situations where multiple producers use another instance of their own class to generate a portion of the end product. In the current state, the individual calls are performed simultaneously. Thus, if the base instance buffers the parts, a race condition that demands particular attention is introduced.

`FifoActivationQueue`

`FifoActivationQueue` pursues the FIFO strategy to store execution requests. This class is the only implementation of the `ActivationQueue` interface. Note that `SimpleScheduler` requires a thread-safe activation queue. The associated code is not illustrated in this work because the concrete methods mainly delegate the emerging invocations to an instance of the JDK's `ConcurrentLinkedQueue`. This class is considered suited to our purposes. It offers thread-safe, wait-free operations to collect the supplied objects.

```
1  public class SimpleScheduler implements Scheduler {
2     private final ActivationQueue queue = new FifoActivationQueue();
3     private Thread thread = null;
4     {...}
5     public void insertRequest(ExecutionRequest request) {
6        if(request == null) {
7           throw new IllegalArgumentException("request must not be null");
8        }
9        synchronized(this) {
10          queue.insertRequest(request);
11          if(thread == null || !thread.isAlive()) {
12             thread = new Thread(this);
13             thread.start();
14    } } } // End of insertRequest
15
16    public void run() {
17       boolean run = true;
18       while(run) {
19          while(queue.hasNextRequest()) {
20             ExecutionRequest request = queue.fetchNextRequest();
21             if(request != null) {
22                request.execute();
23          } }
24          synchronized(this) {
25             if(!queue.hasNextRequest()) {
26                thread = null;
27                return;
28    } } } } // End of run
29
30 }
```

Algorithm 5.2: An excerpt of the implementation of `SimpleScheduler`.

`SimpleScheduler`

`SimpleScheduler` is the standard implementation of the `Scheduler` interface. This component is the central unit for receiving and processing the occurring execution requests. It is also responsible for the allocation and termination of the active object's thread. Algorithm 5.2 shows an extract taken from `SimpleScheduler`'s source code. The following list outlines properties of the type:

- Each instance has fields for the internal activation queue (line 2) and the executing thread (line 3). By default, `FifoActivationQueue` is used for storage of the requests. The `thread` reference is initially set to `null`.
- The utilised queue must be able to cope with concurrent calls. This restriction is supposed to increase the request completion throughput because an `ActivationQueue` implementation can make use of an efficient algorithm to guarantee thread safety.

- `insertRequest` checks the incoming execution requests (lines 6 to 8) and adds them to the scheduler's queue (line 10). The operation then tests if a thread is currently handling requests (line 11) and starts a new one when deemed necessary (lines 12 and 13).
- Requests are executed in the `run` method. Its inner while loop (lines 18 to 22) examines whether the queue holds any remaining requests. If so, they are fetched and invoked one at a time. This part of the method is not coordinated with the inclusion of new requests.
- To avoid potential race conditions, `run` contains a segment (lines 23 to 27) which is synchronised with the request insertion (line 10). This block verifies that the processing thread ends only if no request was added after the last superficial check was made (line 18).

In a nutshell, the chosen approach utilises a performant, thread-safe `ActivationQueue` realisation in order to prevent slowing down the execution of consecutive requests. The final synchronised inspection contributes to the correctness of active objects.

Possible improvements. The displayed code can be altered by allowing multiple clients to add requests to the queue at the same time. In this case, access to the collection has to be secured through a read-write lock. This change improves `insertRequest`'s response time when an active object is heavily used by several threads. However, the adjustment was not incorporated because the projected performance gain is slim. Additionally, this mechanism further complicates the implementation and correctness verification.

Promises

The design described in Chapter 4 uses promises to represent deferred values. Figure 5.1 depicts the whole class hierarchy stemming from this concept. The diagram emphasises overwritten methods by including them in the subtype even though they are already featured in the base type. In the rest of this subsection, each type is explained in greater detail.

`Promise` and `SettablePromise`

The `Promise` interface is the Java translation of the method declarations outlined in Section 4.2. The `SettablePromise` interface expands this type by two new procedures. They allow the completion of the instance.

- `set(V result)` specifies the delivered parameter as the return value of a successful outcome.
- `set(Throwable throwable)` defines the given throwable as the cause of an erroneous completion.
- Invoking a `set` operation of an already resolved promise causes an unchecked exception of the class `InvocationException`.

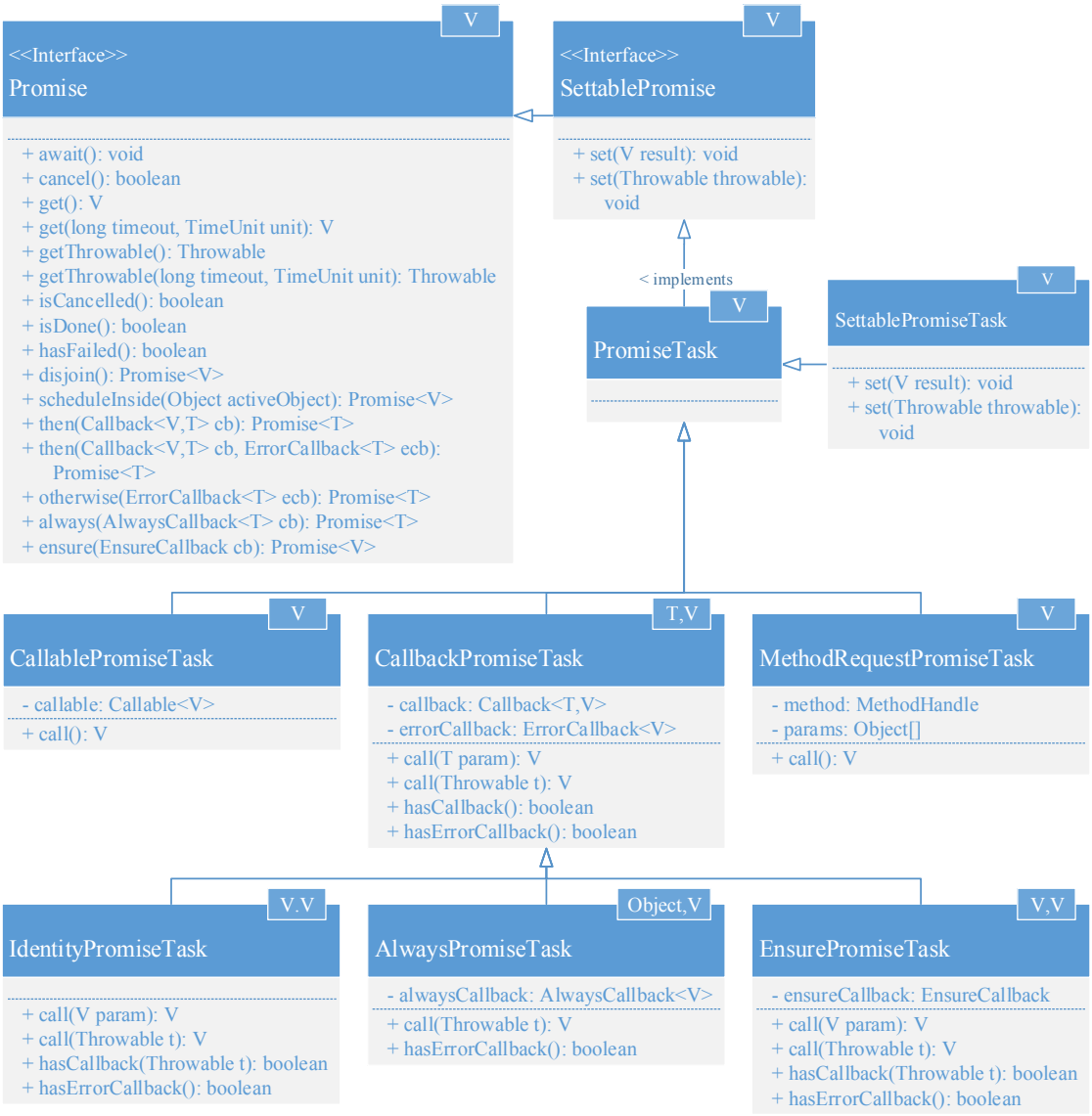

Figure 5.1: The extensive hierarchy of `PromiseTask` implementations.

Settable promises increase the flexibility of the approach because they can be requested via a library function. Typically, the retrieved instance is stored temporarily, returned to the caller and completed at a later moment through one of the `set` operations. This procedure enables the manual use of promises when a computation requires additional active invocations.

`PromiseTask` and `SettablePromiseTask`

`PromiseTask` is the base implementation of the `SettablePromise` interface. This class combines executable tasks with their associated outcomes. It provides the fundamental opera-

```
1  public class PromiseTask<T> implements SettablePromise<T> {
2     protected final Sync sync;
3     {...}
4     public T get() throws InterruptedException, ExecutionException {
5        return this.sync.syncGet();
6     }
7     protected final class Sync extends AbstractQueuedSynchronizer {
8        {...}
9        public T syncGet() throws InterruptedException,
10             ExecutionException {
11          this.await();
12          return this.retrieveValue();
13       }
14       public void await() throws InterruptedException {
15          this.acquireSharedInterruptibly(0);
16       }
17    }
18 }
```

Algorithm 5.3: An excerpt of the implementation of `PromiseTask`.

tions used by the various specialised derivatives. The most interesting aspect in this regard is an inner class named `Sync`. This type extends the JDK's `AbstractQueuedSynchronizer` and makes use of the inherited operations to coordinate simultaneously accessing threads. Because most of `PromiseTask`'s methods have a corresponding operation in `Sync`, they simply delegate the incoming calls. This configuration is based on the `FutureTask` implementation of OpenJDK [1]. `PromiseTask`'s code cannot be included entirely in this thesis due to the extent of the class. Instead, Algorithm 5.3 illustrates the structure using the example of the `get()` function.

- Each task contains a unique `Sync` instance (line 2) in order to forward the basic call (line 5).
- `syncGet` awaits the completion of the promise (line 10) and then returns the final value.
- The call of `acquireSharedInterruptibly` blocks indefinitely until the result is set.

Note that `acquireSharedInterruptibly(int argument)` is obtained by expanding `AbstractQueuedSynchronizer`. This operation tries to acquire a shared lock and only aborts if interrupted. Because of `Sync`'s design, shared locks are only available when the associated promise was completed. The presented segment depends on code parts not covered in the sample for reasons of clarity and comprehensibility. The JavaDocs of `PromiseTask` and the respective JDK classes [2] offer more information on this matter.

Manual use of `SettablePromise`. The `PromiseTask` class is not appropriate for manual use because the completion by a `set` operation does not automatically initiate processing of

the assigned callbacks. Internally, their execution is triggered by other components as soon as the promise is fulfilled. The subtype `SettablePromiseTask` remedies the described shortcoming by overwriting both `set` methods. Their new definitions autonomously commence the traversal of the promise chain after calling the original implementation. Hence, an instance of `SettablePromiseTask` is returned by library functions that yield a settable promise designated for unrestricted use.

`MethodRequestPromiseTask`

This variant of `PromiseTask` is used for active invocations. Every method request holds a `MethodRequestPromiseTask` object to outsource the execution details. Among the member variables of this type are a method handle and an array of parameters. The included `call` operation invokes the handle with the specified set of object parameters, stores the outcome of the invocation and performs the necessary result and exception handling.

`CallablePromiseTask`

A `CallablePromiseTask` addresses the execution of a `Callable` instance. The latter interface is provided by the JDK and has an operation named `call`. This method does not have any arguments. The characteristics of this task type are as follows:

- The constructor of `CallablePromiseTask` receives a callable and assigns it to an instance variable. This parameter is mandatory because it is required by the processing.
- The class offers a single method with the signature `public V call()`. The generic type parameter `V` corresponds to the type parameter of the held `Callable` instance.
- `call()` triggers the execution of the callable's `call` operation and handles the computed result. If an exception is raised during the invocation, the error is propagated to the caller.

Asynchronous invocation service. The behaviour of `CallablePromiseTask` is dictated by the underlying `Callable` object. This relationship allows a developer to wrap directly executable code inside a promise task, even if the implemented mechanism is composed of several separate instructions. In order to use this functionality, the `Active` class provides static utility operations. They receive a callable, package it and return the corresponding promise. The enclosing task is automatically performed either by an independent thread pool or the scheduler of the delivered active object.

`CallbackPromiseTask`

The `CallbackPromiseTask` class concerns the execution of callbacks installed through callback assignment methods. For this purpose, each instance contains fields for a value callback and an error callback. These member variables can be `null` because they are not used by every subclass. We do not go into great detail about the individual kinds of tasks, but rather explain the common functionality and the general processing scheme. The `set` and `call` operations

are responsible for computing and storing the result. Both forms of `call` are newly introduced by `CallbackPromiseTask`. The behaviour of the different methods is explained next.

- `hasCallback()` evaluates to `true` if the `callback` reference is not `null`. Otherwise, `false` is returned.
- `hasErrorCallback()` yields `true` if `errorCallback` points to an actual error callback and `false` if this is not the case.
- `call(T param)` reacts to a successful outcome by invoking `callback` with `param`. The given object is intended to be the value computed by the previous promise. This operation raises an unchecked exception if `callback` does not exist.
- `call(Throwable t)` acts on the throwable raised during the preceding execution by performing the `errorCallback` with the delivered parameter. If `errorCallback` is `null`, the method throws an unchecked exception.
- `set(V result)` and `set(Throwable throwable)` are inherited from the superclass `PromiseTask` and remain unchanged. They are used instead of the `call` operations if the task does not have the necessary callback type to handle the previous result.

Promise chain processing scheme. The behaviour of a task is dictated by the predecessor's outcome. Assuming that the latter was completed successfully, `hasCallback()` must obviously be called to discover whether the task has a value callback. If so, `call(T param)` is invoked. If, on the other hand, `hasCallback()` evaluates to `false`, `set(T param)` is utilised. Because `CallbackPromiseTask` does not hold a reference to its predecessor, this decision needs to be made within the associated execution request.

5.3 Instrumentation Environment

In this section, we explore the surroundings of the programming library and the manner in which active objects are acquired from normal instances. As explained in Section 4.1, the classes are transformed into their active representations through bytecode instrumentation. Figure 5.2 outlines the crucial components of the scheme and their relationships. The primary entity are the *class files*. They are loaded by the *classloading system* and contain the necessary information to recognise types of active objects. The *Java agent* modifies these files and creates new proxy classes on the fly. To this end, it interacts with the *Javassist* library. The latter provides the required functionality to perform a high-level bytecode instrumentation. The next subsections give an explanation of the most important aspects.

Java Virtual Machine

The *Java Virtual Machine* (JVM) is an essential element of the Java Runtime Environment. This component executes the bytecode and provides developers with a wide variety of services. Due to the topic's magnitude, we are not taking an in-depth look at the JVM in this thesis, instead

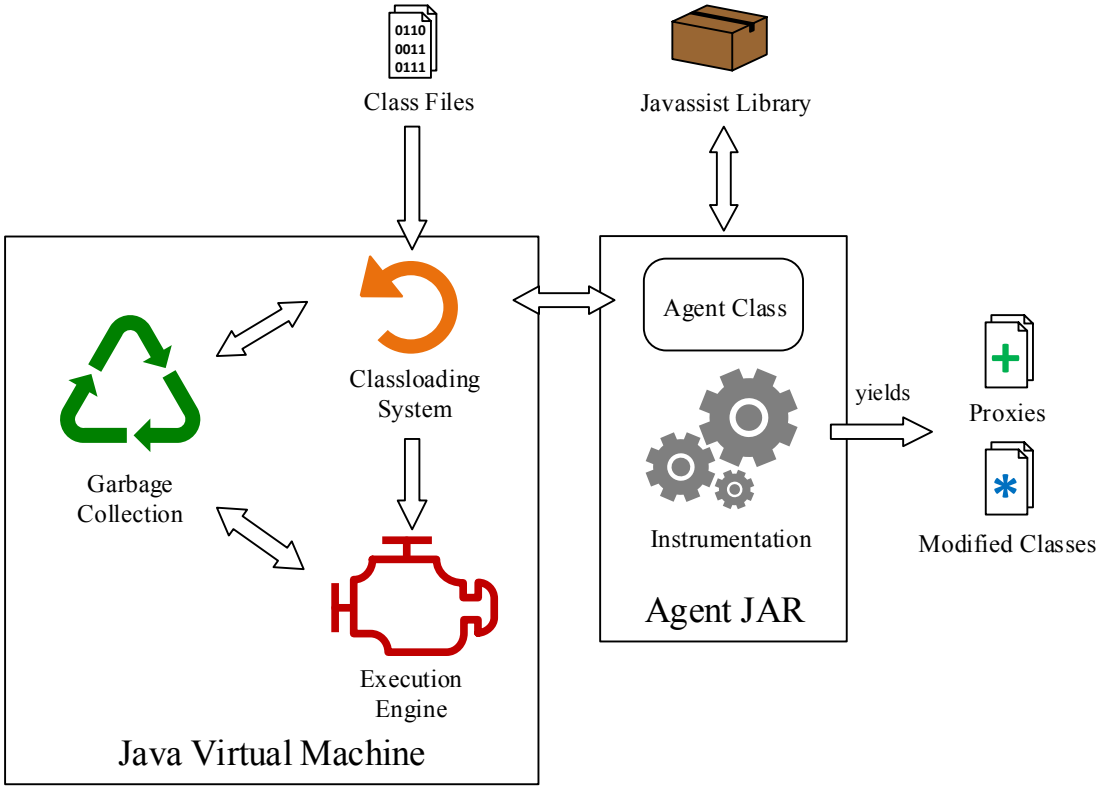

Figure 5.2: Overview of bytecode instrumentation through a Java Agent.

constraining the description to a subarea of the classloading system. As mentioned above, this part loads classes for the runtime environment. Classloading is a complex process, but it is not necessary to understand every detail to comprehend the conducted manipulations.

The `ClassFileTransformer` interface. The JDK's `java.lang.instrument` package offers a range of services with the purpose of instrumenting a program executed by the JVM. The included `ClassFileTransformer` interface is vital for the functional principle of the presented library. This type features a single method with the following signature:

```
byte[] transform(ClassLoader loader, String className,
                 Class<?> classBeingRedefined,
                 ProtectionDomain protectionDomain,
                 byte[] classfileBuffer);
```

An instance of this interface can be registered as a transformer with the runtime system. Consequently, the implemented `transform` procedure is called for each new class definition. This mechanism is the starting point of the presented realisation. The project contains a custom implementation of `ClassFileTransformer`, and all bytecode modifications are triggered by such an invocation of `transform`.

Registration of the transformer. A transformer is enrolled through the `addTransformer` operation of an `Instrumentation` object. This interface is included in the `instrument` package as well. An essential requirement is that the registration occurs before any application classes are loaded, as the instrumentation is otherwise inconsistent. The library must therefore obtain an instance of said type during the startup of the program. This demand is satisfied by the Java agent introduced next.

Java Agent

The *Java agent JAR* is the heart of the set-up. The whole library is wrapped into an archive that comprises the instrumentation, the component and the utility classes. Thus, the only other JAR required in the application's classpath is Javassist. Java agents are passed to the runtime environment by announcing the location of their JAR file to the `java` program through the command-line parameter `-javaagent`. Multiple agents can be handed to the JVM by specifying the expression several times, but the description in this section is restricted to one instance.

Java agents and their manifest. An often neglected file of JARs is the *manifest*. In our case, it has supplementary meaning because its content defines certain characteristics of the agent. The feature of interest is a set of static methods called by the JVM at specific points in an execution's lifecycle. The implementing type is declared in the manifest.

The `Premain-Class` property. We now focus our attention on the `Premain-Class` property. Its value identifies a Java class included in the agent JAR which provides the following operation:

```
public static void premain(String agentArgs, Instrumentation inst);
```

As the name suggests, `premain` is invoked prior to the application's main method. Thus, by implementing this procedure and setting the corresponding property in the manifest, the agent gains access to an `Instrumentation` object before any application classes are loaded by the classloader. This strategy enables the registration of a `ClassFileTransformer` in a way that fulfils the precondition of effective bytecode instrumentation. The same mechanism was also used by Villazòn et al. [52] for initialising HotWave, an aspect-oriented programming framework with support for dynamic weaving.

Proxies and Modified Classes

The analysis and instrumentation can have various non-trivial effects on the involved types. In this respect, the agent has two possibilities to handle a received class.

1. No changes are necessary, because the class neither is active nor uses an active object. As a consequence, the library does not alter any aspect of the given type.

2. The examination encounters the use of an active object within the bytecode. This knowledge involves the creation of a new proxy class if the referenced type has not been treated

yet. Moreover, the analysed class has to be manipulated accordingly, which entails adaptations of the concerned constructor calls.

Therefore, the instrumentation phase yields new or modified classes representing the adjusted application. Certainly, the processed bytecode has to be loaded by the runtime system to have an impact on the execution. The customised class files can also be written to the disk for repeated use. However, any code modifications require a retransformation of the entire application.

Class reloading. The agent cannot apply the typical instrumentation pattern because it depends on services offered by Javassist. Usually, `transform` either returns the manipulated bytecode or `null`. The latter value indicates that no adaptation was performed by the method. The classloader then continues the loading process with the appropriate class definition. On the other hand, the programmed agent loads the modified and created types directly into the context classloader. As a result of this measure, they cannot be retransformed. This limitation is induced by Javassist's functionality. The problem is solved through the use of a custom classloader which supports class reloading. However, such an implementation is not part of the library.

5.4 Instrumentation Approach

A remarkable aspect is the pursued instrumentation strategy. This section addresses the chosen technique and the associated components. An abstract view of the approach is given in Figure 5.3. The enumeration below explains the process in greater detail. The sequential order is expanded by a number of branches and conditional jumps.

1. In the course of loading a class, its bytecode is parsed for occurrences of expressions containing the `new` operator.
2. If an untreated `new` expression is found, continue with Step 3. Otherwise, the workflow is completed in Step 8.
3. The target type of the detected creational statement is searched for the `ActiveObject` annotation. If present, Step 4 is performed. Otherwise, the pending class is rescanned in Step 1.
4. In case the proxy already exists for the annotated type, the discovered expression is adjusted in Step 7. Otherwise, the instrumentation is conducted beginning with Step 5.
5. The servant is adapted and loaded.
6. The proxy is generated and loaded.
7. The encountered servant object creation is substituted with a construction of the corresponding proxy. Thereafter, go to Step 1.
8. The scanned class is loaded and the workflow ends for this particular type.

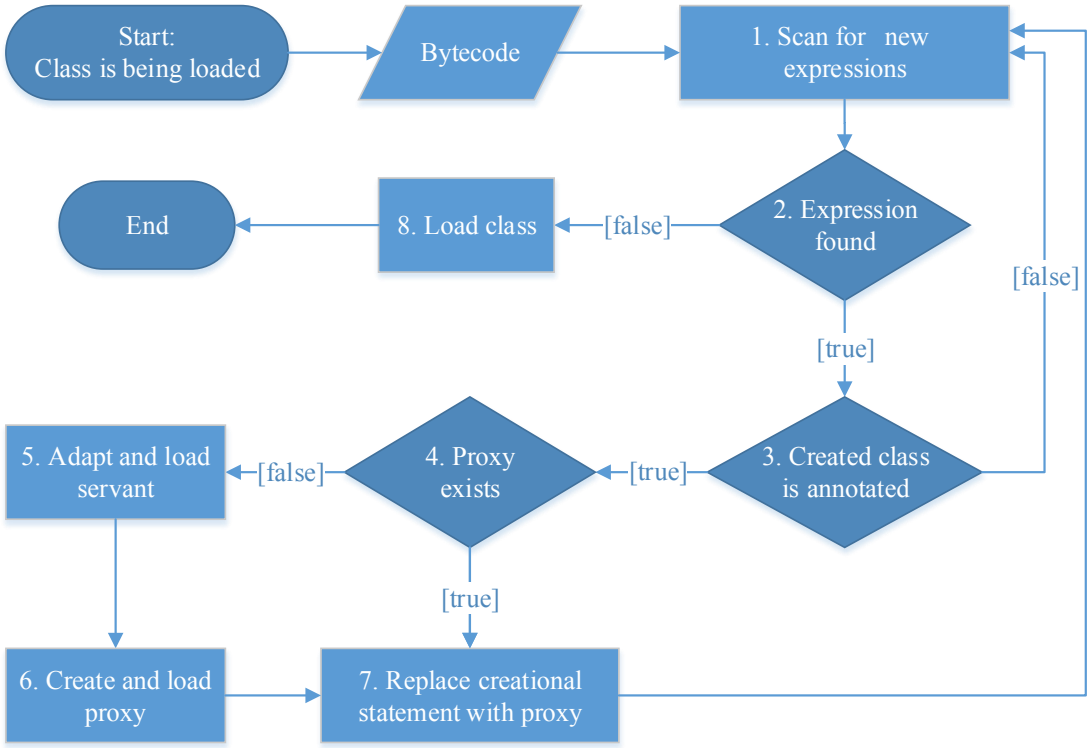

Figure 5.3: A high-level view of the approach chosen for the instrumentation.

To summarise, the above sequence deals with an active object creation in two manners. If the related type has not been handled yet, the agent has to instrument the servant and the proxy prior to the conversion of the creational statement. In the other case, the transformation is limited to the latter task. This procedure is carried out for all loaded application classes. It is also executed for a servant of an active type to ensure full class coverage. This task is included in Process 4 in Figure 5.3.

Proxy generation. As we have seen, the agent generates the proxy classes on demand whenever it encounters a creational statement concerning an active type. This approach is crucial because the chronological order in which the classes are loaded cannot be influenced. In particular, a *first-come, first-served* (FCFS) based instrumentation is insufficient because proxies are created only when the corresponding servant is supplied to `transform`. Imagine there is a class which instantiates an active type and `transform` is called for the former class first. In the case of FCFS, the agent is unaware of the active type and omits the adaptation of the object creation. Since the library cannot reload individual classes, the outcome would be unreliable and error-prone. In conclusion, the instrumentation has to be driven by creational expressions.

The following paragraphs describe the responsibilities and ties of the components that execute the workflow.

ProxyAgent

The entry point of the library is `ProxyAgent`, a custom `ClassFileTransformer` implementation. This class is declared as the agent's `Premain-Class` and features the static `premain` method. `ProxyAgent` registers a new instance of itself in the demonstrated style:

```
public class ProxyAgent implements ClassFileTransformer {
   {...}
   public static void premain(String args, Instrumentation inst) {
      ProxyAgent proxyAgent = new ProxyAgent(inst);
      inst.addTransformer(proxyAgent);
   }
}
```

By the definition of `premain`, the new transformer is added before the classloading system has loaded any application parts. Therefore, `ProxyAgent` fulfils the requirement of an effective instrumentation.

Filtration. As a safety precaution, the provided `transform` method filters the incoming classes by name. This limitation prevents unexpected exceptions stemming from the analysis of system classes. A full list of exclusions is given in the JavaDoc of `ProxyAgent` [2]. For example, all types residing in the `javax` package are ruled out from inspection.

Placement in the workflow. The registered transformer starts the instrumentation by retrieving a changeable meta class object of the received bytecode. The obtained reference is then passed to an `ObjectCreationEditor` instance for further processing. As soon as the editor has fulfilled its mission, the agent returns the bytecode of the class object concerned. `ProxyAgent` is the depicted workflow's starting point and initiates the scanning procedure (cf. Process 1).

ObjectCreationEditor

`ObjectCreationEditor` examines classes for active object creations and incorporates the generated proxies by adjusting these statements. For this reason, the component expands Javassist's `ExprEditor`. The latter type has built-in functionality for parsing class objects. The editor's behaviour is controlled by several predefined methods. They correlate to specific kinds of expressions encountered during processing and can be overwritten in order to introduce the desired changes.

Handling of `new` expressions. `ObjectCreationEditor` analyses and adapts creational statements through the `edit(NewExpr e)` operation. The supplied `NewExpr` instance grants access to information about the concerned expression. The `edit` method acts as follows:

1. The instantiated class is checked for an occurrence of the `ActiveObject` annotation. The editor thereby keeps track of all such types and their associated proxies.

2. If the construction of an unaltered active class is detected, the responsibility of modifying the servant and creating the proxy is delegated to `ProxyGenerator`.

3. Finally, the creation of an active object's servant is always replaced by a constructor call of the related proxy.

Placement in the workflow. The outlined sequence is performed for every object creation present in the given class. When all `NewExpr` instances have been treated, the editor loads the finalised class object and returns control to `ProxyAgent`. With regard to Figure 5.3, `ObjectCreationEditor` conducts processes 1, 3 and 6 and orchestrates 4 and 5.

CodeGenerator

`CodeGenerator` consolidates `static` utility methods, which generate the code strings applied during the instrumentation phase. This class does not perform the modifications itself, but instead provides functionality to other components like `ObjectCreationEditor` and `ProxyGenerator`. They use the services to produce the needed code that is then injected into a class by calling the appropriate compilation operation of one of Javassist's meta objects.

Coupling and dependencies. As a consequence of the mentioned relationships, there is a strong coupling between `CodeGenerator` and the instrumenting classes. Although this solution does not seem optimal, the know-how is outsourced to enable a high-level transformation in the other types with clearly separated competences. At the same time, this procedure eliminates any far-reaching dependencies among these classes.

Placement in the workflow. This thesis does not cover the different features of this type, but the JavaDoc [2] sheds light into their usage. Through the produced code, `CodeGenerator` pays a significant contribution to the adaptation of the servant (Process 4), the proxy construction (Process 5) and the generation of the object creation expressions (Process 6).

ProxyGenerator

`ProxyGenerator` combines the specific transformation capabilities of the library with the aim of modifying servants and creating new proxy classes. This entity offers a high-level interface to conduct the adjustments and is used primarily by `ObjectCreationEditor`. Section 5.5 illustrates a subset of the implemented manipulations.

Functional principle. The layout of the `ProxyGenerator` class entails two crucial characteristics. They are outlined below:

- The supplied operations usually generate a code string by calling a static method of `CodeGenerator` and incorporate the change using Javassist's meta classes. The effect of `ProxyGenerator`'s operations therefore depends on the output delivered by `CodeGenerator`.

- The order of operations is a critical factor for the success of the invocations. Since most of the contained methods alter at least one parameter, calling them with already loaded classes raises an exception. These sorts of defects cause an inconsistent and unreliable instrumentation outcome.

Placement in the workflow. The individual realised functions are not subject of this description. However, the JavaDoc [2] specifies them extensively. `ProxyGenerator` implements the transformation operations that are used to adjust the servant (Process 4), make the proxy (Process 5) and replace the creational statements of the active object (Process 6).

5.5 Applied Transformations

This section is devoted to the instrumentation carried out by the agent's internal components. We concentrate our analysis on a subset of the performed transformations. The particular manipulations are visualised using code samples. These segments are simplified and show source code instead of bytecode. They adapt some parts and names for an easier comprehension. For example, the proxy name suffix is changed from `___ACTIVE_PROXY___` to `Proxy`. The genuine modifications are displayed in `CodeGenerator`'s JavaDoc [2].

Predefinitions. Javassist offers various meta classes, such as `CtClass`, `CtConstructor` and `CtMethod`. They feature a straightforward interface for changing different aspects of the represented subject. Unless noted otherwise, the presented code excerpts comply with the convention listed next:

- `clazz` is an object of `CtClass` and refers to the servant for which the proxy is set up.
- `proxyClazz` is a `CtClass` instance and indicates the proxy.
- `method` is an object of `CtMethod`.

Handling of Final Modifiers

As explained in Section 4.2, a proxy is created by extending the original class and overwriting the active methods. In Java, applying the `final` modifier to these elements prevents the envisioned approach. The bytecode instrumentation has two options to continue in such a situation:

1. The agent ignores the concerned item and does not conduct a manipulation. This option introduces inconsistencies and causes unexpected behaviour if the developer is unaware of the modifier's presence or impact. Displaying a warning alleviates the problem only partially because this kind of message is often overlooked, especially in verbose applications. Additionally, the instrumentation fails altogether under certain circumstances.
2. The alternative is the removal of the modifier from the corresponding element. This possibility accomplishes a successful and consistent transformation of the affected classes.

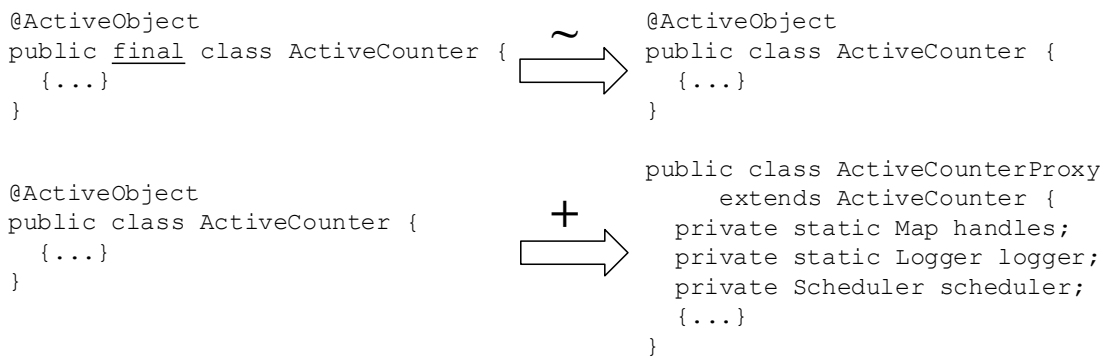

Figure 5.4: A number of code transformations necessary for the creation of the proxy class.

Furthermore, the effects of the modifier are retained at compile-time. However, this adaptation still has negative consequences for certain types of programs. As an example, a running application can check the modifier's existence by means of reflection. This examination yields a false result because `final` was cleared at load-time. The deletion also allows third-parties to extend the adjusted classes during runtime.

For practical reasons, the second variant has been chosen for the library. The transformation's consistency and success are considered more important than the correctness of corner cases. Moreover, a warning message is written to the console whenever a modifier is cleared. This text refers to the affected class or method.

Realisation. The upper part of Figure 5.4 shows `ActiveCounter` before and after the modification. Remember that this action is carried out while loading the class, so the restrictions of `final` are still imposed during compilation. For the purpose of undertaking this change, `CtClass` and `CtMethod` offer the operations `getModifiers` and `setModifiers`. The former method retrieves the modifiers, whereas the latter one sets them in the desired way. The code sample given below removes the `final` modifier from `clazz`:

```
int modifiers = clazz.getModifiers();
if(Modifier.isFinal(modifiers)) {
  if(clazz.isFrozen()) {
    throw new IllegalArgumentException(...);
  }

  Modifier.clear(modifiers, Modifier.FINAL);
  clazz.setModifiers(modifiers);
}
```

1. Since `getModifiers()` returns the modifiers encoded in an integer, a static operation of the `Modifier` class named `isFinal` is used to verify the existence of `final`.

2. `clear` erases a modifier from the passed integer. The affected bit is specified by the second parameter. In this case, `final` is removed from the signature of `clazz`.

3. The resulting value is then handed to `setModifiers(int mod)` in order to define the new modifiers of the class.

Frozen classes. Classes already loaded by the classloading system require special care. This state is indicated by the result of `isFrozen()`. Although a `CtClass` can be *unfrozen*, the adoption of any modifications requires class reloading. As previously mentioned, the agent does not implement this mechanism. Subsequent alterations are therefore not possible and an exception is thrown when a frozen class is encountered by the code.

Set-up of the Proxy Type

As we have already seen, the transparent introduction of a proxy requires the dynamic creation of a class. The new type extends the servant and holds the essential member variables needed by the proxy methods. The lower right-hand side of Figure 5.4 outlines `ActiveCounter`'s proxy class. The generation of the associated base type is easily accomplished by the following segment using Javassist:

```
CtClass proxyClazz = pool.makeClass(clazz.getName()
        + ComponentName.PROXY_CLASS_SUFFIX);
proxyClazz.setSuperclass(clazz);
```

In this context, `pool` refers to a `ClassPool` instance, which can be obtained by calling the static function `ClassPool.getDefault()`. The semantics of the listed code are summarised below:

- The first statement creates an empty class labelled with the delivered string. The proxy's name is composed of the original class's name and a suffix stored inside a constant of the `ComponentName` type.
- The new `proxyClazz` is located in the same package as `clazz` because `getName()` retrieves the fully qualified name in dot notation.
- As an example, `Integer.class.getName()` returns `java.lang.Integer` inside a string. The function `integerClazz.getName()` also evaluates to this value if `integerClazz` is the `CtClass` correlating to `Integer.class`.
- The second command sets the original class as the super class of proxy, so that we have created the needed type.

The proxy class is not loaded automatically by the classloader upon executing `makeClass`. Instead, this happens when one of the `toClass` methods is called on the new `CtClass` instance. This is the final action in the generation of the proxy because each class can only be loaded once and we do not implement a reloadable classloader.

Incorporation of fields. In the next step, the member variables are inserted. Each of them is added by the extract presented below.

```
CtField field = CtField.make(CodeGenerator
     .generateField(fieldType, name, modifier), proxyClazz);
proxyClazz.addField(field);
```

These instructions add a new field to the proxy class. While the functionality of `addField` is straightforward, the usage of `CtField.make(String src, CtClass declaring)` is more complex. The second argument states the class enclosing the field. The `src` parameter expects a variable declaration or definition encoded in a source code text. For instance, the proxy's scheduler field is created by invoking:

```
CtField.make("private at.ajo.scheduler.Scheduler scheduler;",
     proxyClazz);
```

To this end, the `generateField` function produces the appropriate declaration string. For detailed information, see the related JavaDoc [2]. In short, the parameters of `generateField` have the following meaning:

1. `clazz` specifies the field's type.
2. `name` contains the variable's name.
3. `modifier` defines the access modifiers.

In this manner, the proxy receives three fields:

- A scheduler for processing execution requests.
- A class-wide logger to document important events.
- A static collection of method references to the servant's operations.

The last of these class variables is optional. It caches the active method handles in order to improve performance. After the new type is created and equipped with the necessary member variables, the first step of the proxy's generation is completed. The instrumentation proceeds by expanding the class with the essential constructors and proxy methods.

Addition of Constructors to Proxy

In the next phase, the new proxy class is complemented with an adequate set of constructors. As mentioned previously, the instrumentation is driven by creational statements. In particular, the agent has to transparently adapt every encountered allocation of an active object. The replacement expression must construct the underlying servant instance in the originally intended manner. To simplify this procedure, the proxy class has to define every non-`private` constructor offered by the servant. The related members must exhibit the exact same parameters and access modifier. Figure 5.5 illustrates this circumstance by means of `ActiveCounter`. Because the servant has two public constructors, its proxy receives matching ones on the right side.

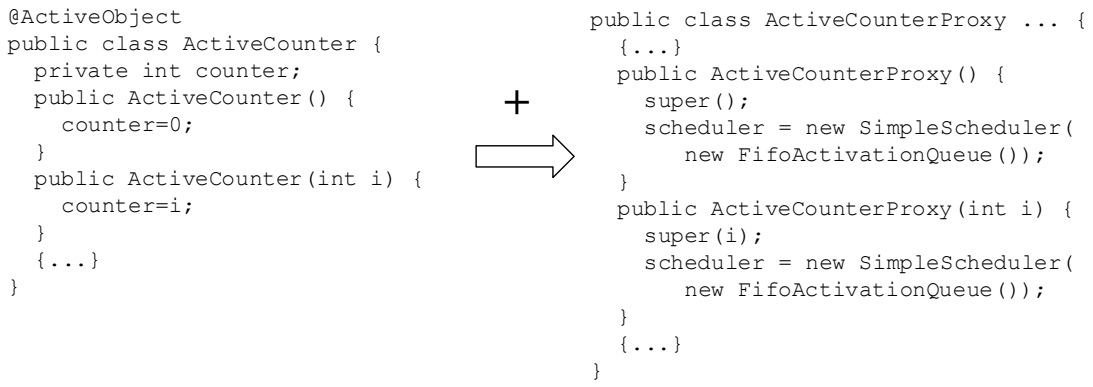

Figure 5.5: Generation of constructors for the proxy of `ActiveCounter`.

Constructors' structure. The body of each introduced constructor has the same basic structure, as shown by the sample code on the right-hand side of Figure 5.5. The implementation needs to complete two general tasks:

1. First `super` is invoked with the passed parameters. This command corresponds to the appropriate servant constructor.
2. Then the internal components of the proxy instance are initialised. In this case, they comprise merely a scheduler object.

Limitations. Java imposes the requirement that a `super` call must be the first instruction inside a constructor's body (cf. Section 8.8.7 of [19]). Therefore, the active object cannot process any method requests before the proxy's constructor has been completed. Since they need to be enqueued into the scheduler instance, the proxy methods would access an uninitialised field. This action raises a `NullPointerException`. Thus, the servant's constructors must not perform an operation which triggers one of the object's active methods.

Handling of `private` constructors. The servant's `private` constructors are a difficult matter. They typically occur in the context of creational patterns. Since these kind of members can be called only within the declaring class, the substituting subclass needs to provide corresponding constructors which are also accessible in the supertype (cf. `public` or default level). Otherwise, the occurrences in the servant cannot be replaced. However, these elements are not covered by the instrumentation. The reasoning of this decision is given in the subsection dealing with the modification of creational statements.

Class initialisation. The *static initialiser block* is another part of the proxy responsible for setting up the active object. During the class loading process, this element is executed in order to prepare the proxy for operation. The contained code conducts the initialisation of the class's static fields. The different tasks are noted below:

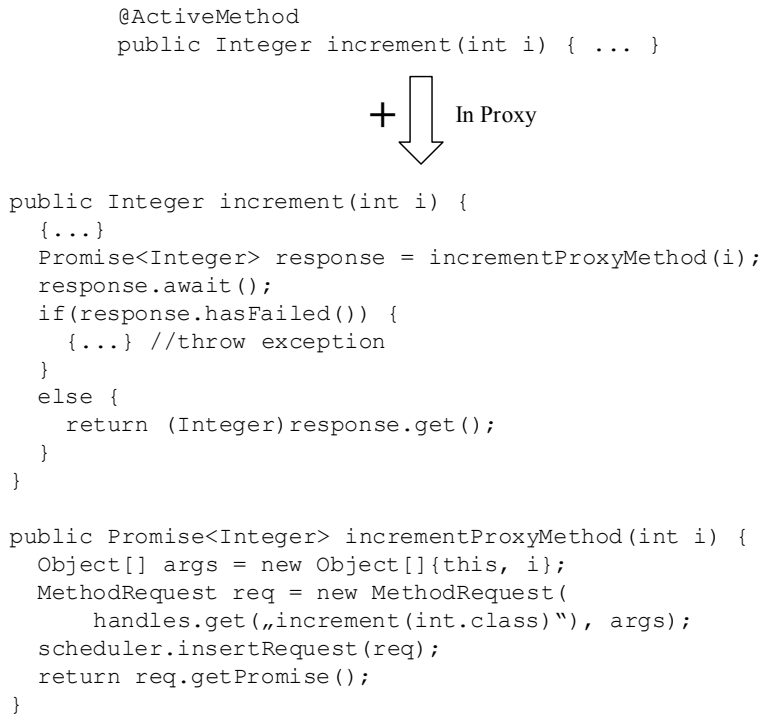

Figure 5.6: Generation of a proxy method from an annotated servant method.

- The aforementioned `Logger` instance is created.
- The block looks up the `MethodHandle` instances of the servant's annotated operations.
- The discovered handles are stored in an easily accessible `Map` instance.

The proxy methods require these handles to create new method requests. This use is demonstrated by `incrementProxyMethod` in Figure 5.6.

Effects. Once all non-private constructors have been integrated, every external constructor call of the servant can be easily replaced with its proper counterpart of the proxy. In this context, external refers to object creations not contained within the code of the servant class definition. In the next phase, the proxy type is augmented by the necessary active method redefinitions.

Addition of Methods to the Proxy

The primary goal of the active object pattern is to separate the method invocation from the associated execution. As yet, the proxy still delegates calls to the servant's operations due to inheritance. This practice clearly violates the desired behaviour since the original method definition is directly accessible. Thus, the proxy needs to incorporate a suitable technique by overwriting the active methods.

Types of proxy methods. To achieve the necessary decoupling, the proposed implementation adds two methods to the proxy for each supported active method. They have the following characteristics:

1. The first method is newly introduced and represents the actual decoupling instrument. This procedure creates a method request, inserts the new instance into the scheduler's activation queue and returns the promise correlating to the result. This method therefore closely resembles the proxy operation of the conventional pattern because both procedures yield a placeholder object.

2. The second method overwrites the servant's operation in order to prevent direct access to the original definition. For this purpose, the functionality of the above procedure is utilised to detach the invocation. If the method is called synchronously by default (cf. `Synchronicity.SynchronousCall`), it awaits the outcome of the retrieved promise and evaluates the result.

Although the former method is declared as `public`, it cannot be called statically because it is not featured in the servant. Hence, asynchronous active invocations of blocking operations are possible only through library functions.

Realisation. The structure of these elements is briefly explained using Figure 5.6. The listed code illustrates their manifestations in the proxy of `ActiveCounter`. At first, we observe the previously undefined `incrementProxyMethod`:

1. The active object and the delivered arguments are repackaged into an array.

2. The corresponding servant method handle is retrieved from the static `handles` map.

3. A `MethodRequest` object is created using the allocated array and the obtained handle.

4. The new request is transferred to the scheduler, and the promise is returned by invoking `getPromise()`.

The details of `increment`'s redefinition are summarised next:

1. The `incrementProxy` operation is used to enqueue a new method request.

2. The promise returned in the preceding step is assigned to a variable.

3. `increment` awaits the completion of the execution because the instrumented method processes a value.

4. If the execution fails, `increment` propagates the occurred exception to the caller.

5. Finally, the result is unwrapped and returned to the client.

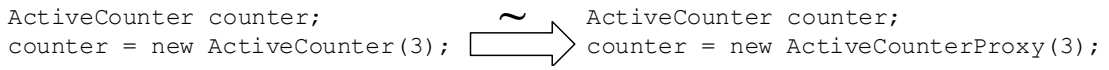

Figure 5.7: Substitution of a constructor invocation to incorporate the proxy class.

Justification for the divided mechanisms. As previously mentioned, the return type of an overwritten servant method cannot be modified in the desired manner because this change violates the substitution principle. Thus, the separation into two methods is required to be able to retrieve the promise directly through library calls. Hence, in order to achieve the desired effect, the proxy has to implement both kinds of operations for each active method.

Modification of Creational Statements

The previously shown bytecode transformations concentrate on the construction of the active object's internal components. Another crucial instrumentation aspect is the incorporation of the active objects into the remaining application. Therefore, proxy objects have to be used instead of servant instances.

Incorporation of proxy objects. The problem is resolved at the object creation level. As explained previously, the proxy provides a suitable implementation of every non-private constructor featured by the servant. Recall that both constructors accept identical parameters. This trait eases the given task significantly. Thus, every servant constructor call can simply be exchanged with an invocation of the proxy's counterpart. The replacement of an `ActiveCounter` creation is demonstrated in Figure 5.7.

Handling of creational patterns. The library cannot apply this transformation to servant constructor calls contained within the servant of the same active object. The reason is that the addition of the proxy introduces a circular dependency. The solution of this problem implies the use of a reloadable classloader. In particular, creational patterns like singletons and factory methods are not possible with the current state of the implementation. Hence, the servant's `private` constructors are not considered by the transformation, since the related invocations cannot be substituted. Aside from these restrictions, the library replaces all static constructor calls in the underlying program.

Dynamic object generation. Reflective object creation poses a particular challenge because the referenced constructor can typically not be decided through static analysis. Furthermore, Javassist does not provide a mechanism for detecting such commands. This deficit is understandable because dynamic constructions are performed by calling specific methods of meta objects. For example, the `Constructor` class located in the JDK's `reflection` package features the `newInstance` method. Hence, these forms of statements cannot be adapted directly since the represented constructor can change depending on the program state. In order to handle them, the agent has to parse every block for calls to known creation methods. The encountered invocations have to be replaced by a custom processing method that checks the target

constructor's class during runtime and creates an instance of the proper type. In case an active object instantiation is recognised, the method chooses the constructors of the proxy over the ones of the servant. This practice does not only complicate the realisation, but also slows down application start-up and degrades execution performance. For these reasons, the agent ignores reflective object creations altogether.

CHAPTER 6

Evaluation Environment

This chapter lays the foundation for the evaluation illustrated in Chapter 7. To that end, a multi-threaded application is implemented by means of conventional Java concurrency and through the use of the library proposed in Chapter 5. First of all, the main scenario is briefly introduced in Section 6.1. Section 6.2 treats the comparison criteria applied to the individual realisations. All dimensions include a detailed explanation as well as reasoning of their importance. The principles of the analysed implementations are explained in Section 6.3. This segment also covers the chosen parameter settings.

6.1 Concurrent Scenario

The benchmark used as the basis of the evaluation in Chapter 7 is an advanced producer-consumer problem. Figure 6.1 depicts the participating entities and their relationships. The class diagram includes various versions of the same method separated by two vertical bars, for example the `order` procedure of `Supplier`. The concerned operations represent alternatives. This unusual detail indicates that a realisation must cover exactly one of the methods in question but not both.

Components

The following list gives an overview of the important components. They are presented more thoroughly in Appendix A.

- `Item` and `Part` objects are the key elements of the system, and they are passed between the interacting components. The scenario supports different item types (cf. `ItemType`), all of which have a distinct composition of the existing part types (cf. `PartType`).
- The `Shop` class markets items to customers. Therefore, it features a function for querying the offered types. Items cannot only be sold but also refunded. Shops request new items from warehouses.

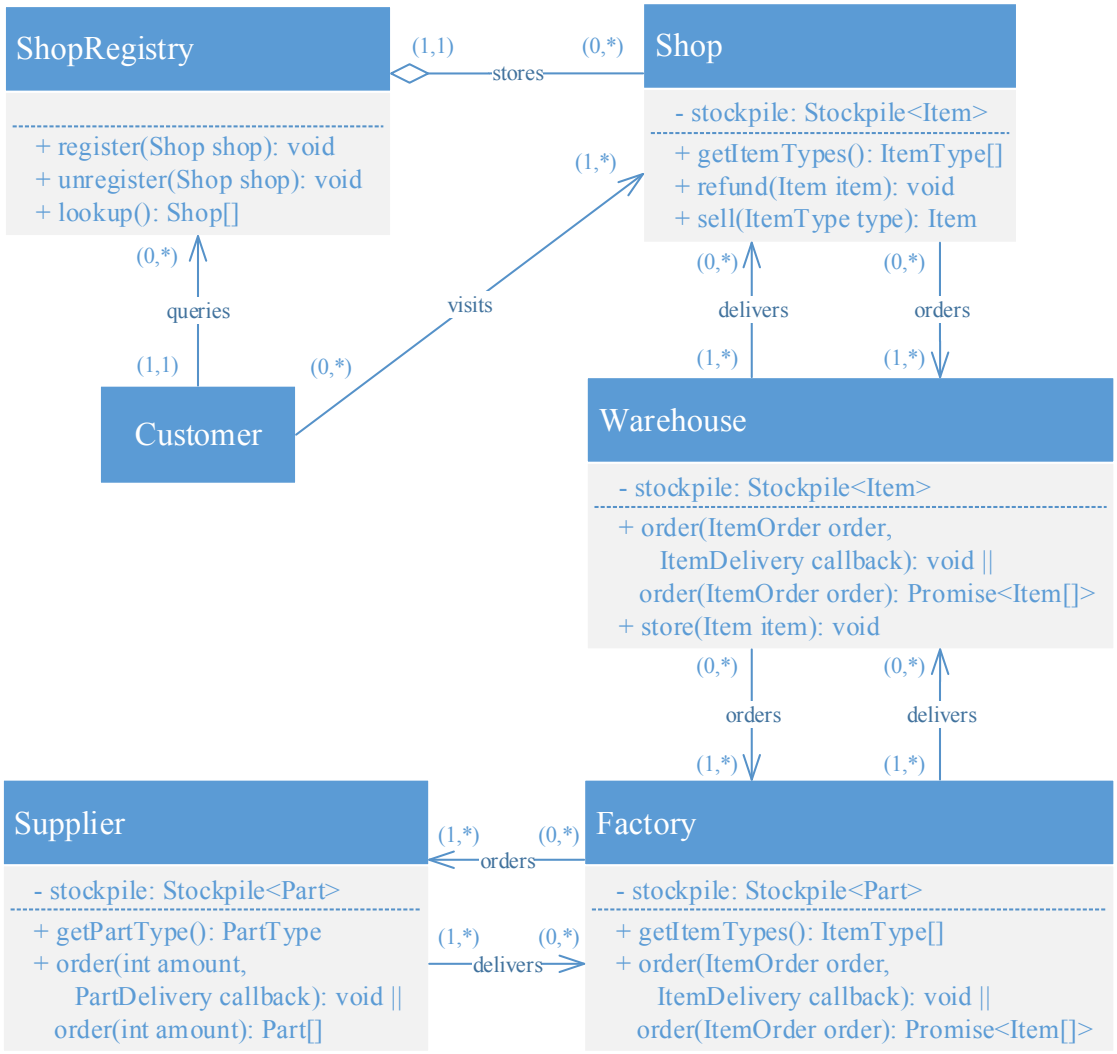

Figure 6.1: The class diagram of the shop example.

- Instances of `ShopRegistry` serve as rendezvous points. They provide registration and deregistration operations for shops and enable the discovery of active vendors.
- A `Customer` thread finds shops by accessing a registry and selects one of them to buy an item. The latter is refunded to the original seller if certain conditions are met.
- A `Supplier` instance creates exactly one particular kind of part. This class exhibits methods for querying the generated `PartType` and requesting a given amount of articles.
- A `Factory` object combines parts to items. It provides functions for inquiring the producible types and ordering items. To obtain the required parts for manufacturing, a factory directly communicates with suppliers.

- A `Warehouse` buffers all kinds of items. Retailers can acquire these articles and return individual ones. This component replenishes its reserves through the existing factories.

Finite storage capacity. A crucial characteristic is that the resources of shops, warehouses, factories and suppliers are limited. Hence, in order to refill their inventory, they need to interact with other entities. In this respect, the `Stockpile` object is a critical element of the aforementioned classes. It has two thresholds. The upper value defines the maximum amount the stockpile can accommodate of each type. On the other hand, the lower value specifies when new goods have to be fetched. More precisely, all administered types are reordered once one of them falls below this limit.

Execution Parameters

The outlined system can be customised through a variety of settings. The most interesting adjustments are summarised below:

- The available kinds of parts (cf. `PartType`)
- The offered sorts of items (cf. `ItemType`) and their part compositions
- The production time of a single item or part
- The number of suppliers, factories, warehouses, shops and registries
- The dynamics of the components. For instance, they can come and go over time.
- The stockpiles' upper and lower limits
- The customers' refund rate regarding the purchased items

6.2 Evaluation Criteria

This section gives an overview of the employed evaluation criteria. They are solely used to assess the applications but not the library classes. These metrics include general ones which may be applied to a wide range of software solutions, as well as others specific to the scenario presented in Section 6.1. In this regard, rating a given value is usually a difficult matter. This task is eased by comparing multiple approaches. The next subsections explain the idea behind the individual criteria and the reasons for their inclusion in the analysis.

Lines of Code

The *lines of code* (LOC) states the number of program lines a realisation utilises. The result suggests the effort of implementing the approaches. Each line is counted once, even if it consists of multiple instructions.

Source lines of code. The LOC metric comes in several variants. In this thesis, we only consider lines containing program statements. Comments and blank lines are excluded from the measurement. This restriction is often labelled *source lines of code* (SLOC). If not mentioned explicitly, LOC means SLOC in the rest of this thesis.

Forms. The default metric covers the entire application or complete classes. However, the different implementations of the scenario have a common code base to ensure a fair assessment. Since the generated value is influenced greatly be these shared resources, its expressiveness is limited. The evaluation therefore incorporates supplementary, extended criteria. They target the synchronisation code because this area shows the highest number of differences. The next chapter explains the applied specialisations in detail.

Bytecode Size

The bytecode size determines the amount of disk space the compiled class files occupy. The metric can cover an arbitrary quantity of types. Modern systems usually possess large amounts of hard drive space. Nevertheless, this characteristic has more impact than the pure storage requirements and is therefore considered in this thesis.

Forms. Similar to the LOC, the measurement of an implementation's base size provides only part of the big picture. For this reason, additional key figures are computed. They address the classes synchronising the core components. Another interesting aspect is the instrumented bytecode. It illustrates the extent of the automatically added boilerplate code in relation to the actual implementation. This information, together with the determined LOC figures, suggests the development effort saved by the deployment of the library. The concrete metrics are described in Chapter 6.

Code Characteristics

This category compares the implementations' source code. Hence, it does not calculate a value but instead discusses development aspects. The main focus is on the interactions between the different entities. The classes containing the fundamental functionality are not considered because they are shared among all programs. The analysis also addresses the structure and synchronisation of the components. The demonstrated code is examined in terms of its complexity and potential faults. The identified characteristics are used for comparing the different approaches and to derive more general arguments about the necessary programming effort.

Customer Timing

This metric registers the time span a particular application needs to serve a predetermined number of customers. The corresponding threads are started in succession, separated only by a fixed delay. The measuring entity awaits the completion of all initiated clients before stopping the timer. As a consequence, the result indicates the manner in which the realisation handles a burst of requests. Lower values imply better performance.

Adaptations. The behaviour of the execution can be altered by adapting the delay between starting threads. A wait time of zero achieves simultaneous processing of clients. Moreover, the total number of customers is modifiable. The produced outcome can be used as the basis of additional figures, such as the average completion time or the count of items sold per minute.

Customer Throughput

The throughput criterion evaluates the number of customers the application can successfully process in a predetermined period. For this purpose, a constant pool of concurrently querying threads is allocated. A new client is activated whenever a running one terminates within the defined time window. Hence, the metric states the implementation's capability of dealing with a given number of consumers. A higher value indicates better performance for the selected parameters.

Adaptations. This criterion covers multiple situations by changing the number of simultaneous threads. Additionally, expanding the prescribed time frame increases the calculated metric's meaningfulness. As with previous criteria, further metrics can be computed from the produced value to gain a deeper understanding of the implementation's efficiency and scalability.

Component Response Time

The response time reveals the speed at which an entity is able to answer incoming requests. The recorded data helps identify bottlenecks and compare the quality of the different realisations. Lower values represent a better score here.

Adaptations. The expressiveness of a total sum is inadequate because the number of calls typically fluctuates. The result is therefore specified on a per method invocation basis. This metric is calculated by dividing the added response times of all requests by the number of calls. Consequently, a low value implies that a single invocation usually returns after a short duration. A component's result can thereby be determined per instance or class.

Memory Consumption

An important performance aspect is the amount of space an application consumes at runtime. High memory requirements do not only impair efficiency but can also prevent normal termination of the execution. Thus, an implementation with reduced needs has an advantage.

Measuring method. In the case of Java, the JVM allocates a certain amount of main memory, which is virtualised and managed (cf. garbage collection) on behalf of the processed application. For this reason, the internally used memory cannot be inferred directly from the information provided by the operating system. To remedy this shortcoming, the JDK offers utility classes for the purpose of determining the real demand. They enable access to the latest statistics regarding memory consumption, such as the current free space in the JVM.

Aggregation. A sole metering point does not provide many details about long-term needs. The data must therefore to be collected throughout the program run to compute adequate aggregate values. Among the interesting metrics are the mean real-space consumption and the maximum memory usage. This practice accomplishes an accurate measure and enables a meaningful assessment of the realisation's behaviour. The gained information is consolidated and visualised to extract additional attributes.

6.3 Principles of the Implementations

To obtain a consistent and comparable result, all approaches are tested under the same conditions. Section 6.1 described a set of parameters to control the execution of the metered application. The selected settings should support the deduction of suitable performance characteristics. In this sense, complex and opaque decisions must be avoided because they are shared between all implementations and distort the end result. An example of these undesired influences is a load-balancing algorithm for order placement. The terms chosen for the evaluation are summarised below. They are divided into configurations of articles, components and execution parameters.

Articles.

- Ten types of parts
- Three kinds of items
- Five part types are required by all forms of items.
- Each of the remaining part types is needed by one item type exclusively.
- The production time of one `Item` or `Part` object is zero.

Components.

- In total, the system has ten suppliers, exactly one for every `PartType`.
- One factory assembles products regarding all `ItemType` values.
- A sole warehouse functions as the central buffer.
- A single shop serves the incoming `Customer` threads.
- One unique shop registry connects customers to the shop.
- During the execution, all previously mentioned components stay active. In particular, there is no coming and going of any entity.
- Each stockpile has a maximum capacity of ten and a minimum limit of three (cf. per type).
- Items are never returned by customers (cf. `refund`).

Program runs.

- Timing measurements are performed with 10,000 customers. The average of five executions is calculated and used as the basis of discussion.
- The throughput of the programs is metered with 1,000 simultaneous customers, with each run taking one minute. The mean of five runs is condensed.

We now focus our attention on high-level implementation aspects. Initially, the different manners of realising a single component are addressed. Thereafter, the concrete applications are explained in greater detail.

Component Variants

This subsection outlines various ways of implementing the components. The following paragraphs present three approaches:

1. Conventional threads and explicit callbacks
2. Active objects and explicit callbacks
3. Active objects and implicit callbacks

Distinction. A key differentiator is the ordering and delivery procedure. Classes of options 1 and 2 are exchangeable due to their identical interaction pattern. In contrast, Option 3 is not directly compatible with the other two mechanisms. Hence, collaboration among the mismatching styles requires special treatment. This feature is not considered here.

Conventional Approach with Explicit Callbacks

The standard strategy is built on components with hand-crafted synchronisation and explicit callback operations. This variant serves as the baseline for comparison. Figure 6.2 depicts the situation by showing the interactions between a shop and a warehouse.

Realisation. The component class extends `Runnable`. As a result, each instance has its own thread for placing orders. The structure of this process is noted below:

1. A client invokes a method that accesses the entity's inventory.
2. Before the operation returns, it checks whether any product types are required. If so, it awakes the instance's thread.
3. The thread determines the missing quantities and issues an order with one or more vendors.
4. The distributor invokes the callback function of the recipient once the whole delivery is created. This operation inserts the supplied articles into the stock.

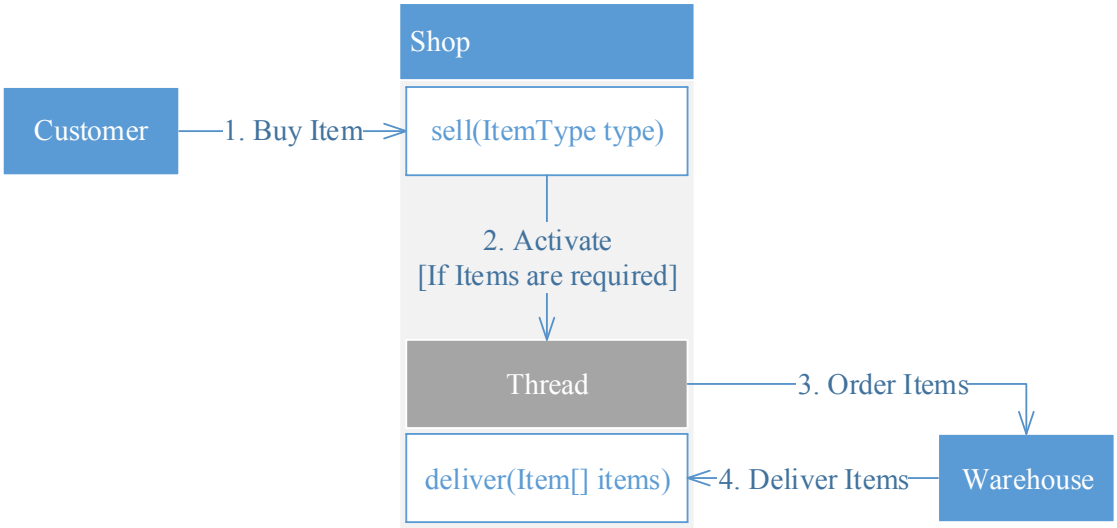

Figure 6.2: Outline of a conventional implementation of the `Shop` class.

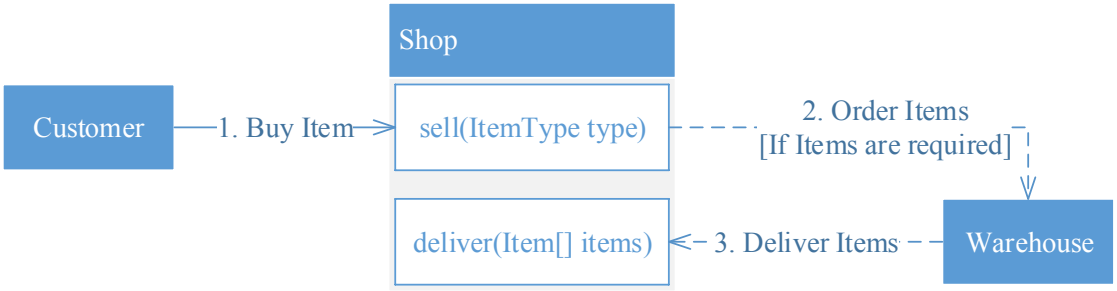

Figure 6.3: The active `Shop` class based on explicit callbacks.

Instead of using active objects, the critical sections are directly synchronised through built-in concurrency constructs (cf. the `synchronized` keyword). This manual coordination is necessary because the operations and the triggered thread try to access the inventory simultaneously. In Figure 6.2, Transition 2 of the `deliver` method has been omitted for clarity.

Active Object Approach with Explicit Callbacks

The next approach uses features offered by the library. The components are implemented as active objects and all critical sections are enclosed by active methods. The explicit callback interaction remains the same. Figure 6.3 illustrates the collaboration by the example of shop.

Realisation. In this context, both `sell` and `deliver` are active methods. The steps of the procedure are:

1. An operation using the private stockpile is executed. (cf. `sell`)

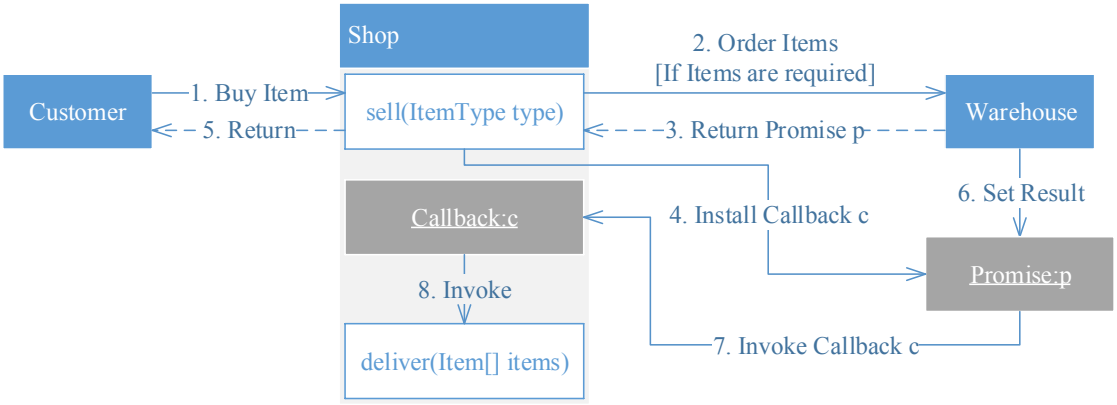

Figure 6.4: The active `Shop` class based on promises.

2. Before the method terminates, it examines the supply and places an order with the sellers if necessary. The latter task is performed asynchronously.

3. As soon as the demanded products are collected, the distributor calls the active object's `deliver` operation asynchronously.

Since only one active method runs at any given time, no synchronisation primitives are needed and the internal inventory can be accessed freely. In this constellation, the asynchronous call structure is vital because it prevents the occurrence of deadlocks. Again, Figure 6.3 does not cover Action 2 of the `deliver` method to streamline the presented diagram.

Active Object Approach with Promise and Implicit Callbacks

The last variation mentioned here is also constructed from active objects. However, it relies on promises and callback assignments instead of the explicit pattern used previously. Figure 6.4 depicts the corresponding `Shop` type.

Realisation. The individual phases of the approach are listed below:

1. The process starts with the invocation of a method that utilises the product storage.

2. Before the operation ends, it places an order if the amount of any article type has fallen below the lower threshold.

3. Next, the promise associated with the request is retrieved.

4. The method proceeds by installing a callback for handling the expected delivery.

5. The operation terminates and returns control to the client.

6. Once the shipment is arranged, the vendor sets the result of the corresponding promise.

7. The callback assigned in Step 4 is automatically invoked upon completion of the promise.

8. Finally, the callback inserts the provided items into the inventory.

The above procedure is only possible if the distributor's `order` method returns a promise or an actual array reference.

Data hiding. In contrast to the previous approaches, the receiving method does not have to be publicly accessible. Therefore, it cannot erroneously be directly invoked by other classes.

Applications

Several implementations were created from the above patterns in order to deduce statements of the characteristics entailed by the library's functionality. The variants differ in the way the `Shop`, `ShopRegistry`, `Warehouse`, `Factory` and `Supplier` components are implemented. The layout of the derived four applications is described next:

1. The conventional program applies the traditional technique with explicit callbacks to all components. This version serves primarily as the reference for the other realisations.

2. The first application of active objects is based entirely on explicit interactions. It is mainly used to assess the ease of use and overhead of the basic library services.

3. The second active object application handles results through promises. It demonstrates the development of implicit callbacks and highlights the performance of their processing.

4. The last active object program pursues a hybrid strategy by using implicit callbacks only for the `Supplier` component. The remaining entities expect to deliver the goods via a callback object.

Reasons for the hybrid form. The hybrid approach strikes a balance between the two presented methodologies and is intended to show the negative effects of the implicit style. The latter changes the programming of the concerned types, especially if they cannot respond to an order immediately. In such a case, their methods have to return a promise because the actual result is not ready in due time. This behaviour needs additional management in the corresponding classes. Furthermore, the use of these kinds of operations affects the performance because they require extended treatment by the library.

Although possible, the cooperation of conventional and active object-based components is not considered by an application. The next chapter presents the outcome of the evaluation.

CHAPTER 7

Evaluation Results and Comparison

This chapter assesses the characteristics of the implementation outlined in Chapter 5 using the applications and criteria explained in Chapter 6. Although the result of each mentioned category is discussed in an individual section, the descriptions aim to establish connections between the diverse measurements. Additionally, the outcomes are justified where possible. Finally, the findings and implications are summarised in Section 7.8.

7.1 Code Characteristics

This part examines the code from the developer's view point. The next paragraphs present and compare remarkable properties of all applications. The analysis focuses on the core components and their interactions since the primary differences of the programs lie in these aspects.

Adjustments. The illustrated samples are adapted to obtain a clearer picture of the emphasised details. Therefore, dispensable segments are removed by consolidating and reducing the underlying classes. These simplifications are necessary since the majority of the functionality is shared among the approaches to avoid repetition and guarantee a fair evaluation.

Structure of Components

The most basic distinction between the concepts is the component layout. It comprises the class declaration, method signatures and the used annotations. These features are now explored in depth together with other fundamental properties.

Active Objects

First we take a look at the proposed concurrency scheme. The following excerpt gives an overview of `ActiveShop`, the implementation of `Shop` based on explicit callbacks. The expla-

nation does also apply to the other active object realisations. They are not mentioned separately since they have an almost identical structure.

```
@ActiveObject
public class ActiveShop implements ItemDelivery, ... {
   {...}

   public ItemType[] getItemTypes() { ... }

   @ActiveMethod
   public Item sell(ItemType type) throws ... { ... }

   @ActiveMethod(Synchronicity.Message)
   public void deliver(Item[] items) { ... }
}
```

Class definition. Other than being marked with `ActiveObject`, the type does not have to meet any demands.

Reliability. Each method has to be self-contained because the communication is mostly unpredictable. In particular, all interactions must be possible in any situation without causing an unexpected or erroneous successor state.

Method settings. Every operation is either active (cf. `ActiveMethod`) or passive. As previously mentioned, the latter form does not feature any annotation and allows multiple calls to run concurrently, not just among themselves but also alongside a single active invocation. Hence, this variant must be employed cautiously. Another essential choice is the default call mode of active methods (cf. `Synchronicity`).

Consequences. The above aspects require careful consideration during the design. If a class or any of its operations is marked incorrectly, the fault can lead to unexpected behaviour. Moreover, the configuration needs to be examined whenever a component is modified. On the positive side, the central decisions are visible by looking at the fundamental class structure.

Conventional

In contrast, observe the architecture of `StandardShop`, the `Shop` realisation of the traditional approach:

```
public class StandardShop implements Runnable, ItemDelivery, ... {
   {...}
   public ItemType[] getItemTypes() { ... }
   public Item sell(ItemType type) throws ... { ... }
   public void deliver(Item[] items) { ... }
   public void run() { ... }
}
```

Class definition. The ordinary alternative does not exhibit any annotations by default. Instead, the class implements `Runnable` because the instances use a thread for ordering items. This behaviour could be extracted into an inner class or a `FutureTask` but was kept in this fashion to highlight the point.

Methods. The individual methods do not necessarily have to be categorised in the design process. However, if the original concept proves to be unfeasible, the core structure needs to be adapted during the development. Furthermore, the impact of the chosen solution is not apparent from the basic class layout. In this case, the code and documentation have to be studied thoroughly.

Synchronisation of Critical Sections

A crucial facet of parallel programming are critical sections and their protection. These characteristics are now considered in the context of the different methodologies.

Conventional

The traditional approach is synchronised explicitly, and an entity can only function properly if all critical sections are enclosed by appropriate concurrency constructs. As a consequence, insufficient coverage can cause a variety of flaws. Some of them are visualised by means of `StandardShop`'s `refund` operation.

```
public void refund(Item item) {
   synchronized(this.stockpile) {
      if(!stockpile.containsType(item.getItemType()) ||
                stockpile.isFull(item.getItemType())) {
         this.centralWarehouse.store(item);
      }
      else {
         this.stockpile.add(item.getItemType(), item);
      }
   }
}
```

Defects. In the above excerpt, the held stockpile is guarded by a `synchronized` block. This endeavour is effective when all statements using the instance are controlled in the same way. However, if one of them is secured incorrectly, race conditions can occur. The list below mentions three problematic situations:

- *Unrestricted usage.* Shortly after `isFull` returned `false`, an unmonitored segment fills up the supply of the corresponding `ItemType` entirely. Subsequently, `add` has to discard the item. In stricter use cases, this call raises a runtime exception not caught by `refund`.
- *Varying monitors.* One part of the code accessing the stockpile is wrongly monitored by a different object. Hence, the efforts are rendered completely ineffective, and they give

a false sense of security. This kind of oversight has similar effects as explained in the previous paragraph.

- *Monitor replacement.* If the `stockpile` reference is exchanged during execution, any simultaneously running `refund` call is endangered. Due to the assignment, the old value still controls the `synchronized` statement, whereas the new one is used inside the block.

The classical technique is considered error-prone for the above reasons. Although a type can implement only fully `synchronized` methods to reduce the attack surface, this strategy is not safe from mistakes regarding state sharing (cf. aliasing).

Active Objects

Consider the code of `ActiveShop`'s `refund` operation given below:

```
@ActiveMethod(Synchronicity.Message)
public void refund(Item item) {
   if(!stockpile.containsType(item.getItemType())
         || stockpile.isFull(item.getItemType())) {
      this.centralWarehouse.store(item);
   }
   else {
      this.stockpile.add(item.getItemType(), item);
   }
}
```

As illustrated by the example, critical sections do not need special treatment, because only one active method is running at a time. Thus, the development phase concentrates on the translation of the design instead of the correct coordination. The programmer still must ensure a single resource is not used by multiple objects. Nevertheless, this simplification is a significant improvement over the traditional approach.

Defects. In contrast to the aforementioned bugs, problems regarding the suggested concept originate from more fundamental sources and are therefore simpler to identify.

- Race conditions correlate to a missing annotation in the class definition or are caused by unwanted state sharing (cf. a function returning an internal reference).
- Deadlocks occur when several active objects synchronously call methods of each other in a circular pattern.

Typical for active object communication are deadlocks that happen deterministically. They can be traced by observing the interactions between the involved components (cf. the chosen `Synchronicity` settings). In the conventional methodology, this problem is caused by resource conflicts triggered by specific events. The latter may not be reproducible in a reliable way without modifying the code. In this case, the debugging task is aggravated by the necessary changes. As a result, active objects have an important advantage over traditional realisations with respect to the development and error detection.

The Process of Ordering Articles

A major distinction between the implementations is the manner in which orders are placed. The ordinary approach uses a separate thread for this purpose, whereas the active object variants rely on asynchronous invocations. Although this call mode is also possible by using `FutureTask`, this practice requires additional effort to program the thread conducting the operation.

Conventional

The source code stated hereafter outlines the situation of the `StandardShop` class. The structure of the other components is similar.

```
public class StandardShop implements Runnable, ... {
   {...}
   public Item sell(ItemType type) throws ... {
      Item item = this.retrieveItem(type);
      boolean hasNeededTypes = false;
      synchronized(this.stockpile) {
         hasNeededTypes = this.stockpile.hasNeededTypes();
      }
      if(hasNeededTypes) {
         synchronized(this) {
            this.notify();
         }
      }
      {...}
      return item;
   }
   public void run() {
      while(this.run) {
         ItemOrder order = null;
         synchronized(this.stockpile) {
            if(this.stockpile.hasNeededTypes()) {
               order = this.createOrder();
            }
         }
         if(order != null) {
            this.placeOrder(order);
         } else {
            try {
               synchronized(this)
                  this.wait();
            }
         } catch (InterruptedException e) { }
      }
   }
}
```

This excerpt is fairly extensive, so we focus on its meaning before discussing the involved issues.

Sales. `StandardShop`'s `sell` method handles a single purchase by carrying out the following actions:

1. An attempt is made to acquire the desired item type (cf. `retrieveItem`).
2. `hasNeededTypes` is called to test if any `ItemType` has fallen below the stockpile's lower limit.
3. If the previous step yields `true`, the internal thread is notified.
4. Finally, either the item or, if no instance could be obtained, `null` is returned.

The task of requesting the products is outsourced, so `sell` can complete as fast as possible. This alternative supports a low response time since the termination occurs earlier and is not delayed until the order is processed by the warehouse.

Thread. As a consequence of the above sequence, the thread is awakened when articles are required. The corresponding `run` procedure performs the noted instructions as a reaction to this demand.

1. The stockpile is checked again by invoking `hasNeededTypes`.
2. If the preceding query returns `true`, `createOrder` is used to generate an item order (cf. `ItemOrder`). Otherwise, the thread goes to sleep and waits for the next notification.
3. The request is transferred to the warehouse through the `placeOrder` operation.
4. Once the invocation terminates, continue with Step 1.

The `synchronized` statement should not include the `placeOrder` call. If this aspect is not taken into account, the performance is degraded because no other sale can be carried out simultaneously.

Difficulties. The main challenge of the above code is its complexity. The points leading to this impression are summarised below:

- Access to the stockpile has to be synchronised manually. Therefore, errors in the form of race conditions or deadlocks are easily introduced.
- Close attention has to be paid to the `synchronized` blocks. Their positioning and scope has to consider efficiency and correctness concerns.
- The code is rather obscure and confusing as it includes several peculiarities not recognised at first glance.
- The non-linear execution path further complicates the comprehension of the situational behaviour.

Active Objects

The following segment demonstrates the corresponding implementation of `ActiveShop`.

```
public class ActiveShop implements ... {
   {...}
   @ActiveMethod
   public Item sell(ItemType type) throws ... {
      Item item = this.retrieveItem(type);
      if(this.stockpile.hasNeededTypes()) {
         ItemOrder order = this.createOrder();
         this.placeOrder(order);
      }
      {...}
      return item;
   }
}
```

Sales. `ActiveShop`'s `sell` operation has a similar but overall simpler layout when compared to the conventional counterpart.

- No specific synchronisation is needed because at most only one active invocation is executed at once.
- No separate thread is used. Instead, `placeOrder` submits the request asynchronously. Hence, the procedures assume a linear style which is easier to understand.
- The time and effort necessary to develop the `run` method are saved by the underlying threading model.

The Delivery of Articles

As explained in Section 6.3, the results are handled by callbacks. In this thesis, we only consider the active object applications in greater detail. The traditional approach is similar to the explicit concept but involves custom synchronisation. This circumstance was already discussed above and is not of additional interest to this description.

Explicit Callbacks

First we address the invocation pattern relying on the callback interfaces `PartDelivery` and `ItemDelivery`.

Delivery. The following sample depicts the implementation of `ActiveFactory`:

```
@ActiveObject public class ActiveFactory
      implements PartDelivery, ... {
   {...}
   @ActiveMethod(Synchronicity.Message)
   public void deliver(Part[] parts) {
```

```
        this.processDelivery(parts);
    }
}
```

The shipments are sent to the `deliver` method. It is defined `public` by the `PartDelivery` interface so as to be accessible to other classes. The transmitted parts are inserted into the stock by invoking `processDelivery`, a base operation shared among all realisations.

Ordering. A request can be placed with the command `supplier.order(partCount, this)`. The meaning of the concerned variables is as follows:

- The employed supplier features the method:
  `void order(Integer amount, PartDelivery callback)`
- `partCount` contains a positive integer.
- `this` passes the current object as the callback (cf. `PartDelivery`).

Implicit Callbacks

We now consider the use of the library's promise capabilities.

Delivery. This kind of approach processes the delivery in a different manner. Observe the segment:

```
@ActiveObject public class CallbackFactory ... {
    {...}
    private final Callback<Part[],Void> processCallback =
            (Part[] parts) -> { processDelivery(parts); return null; };
}
```

Instead of defining a `public` method, a `Callback` instance variable is initialised through a lambda expression. The latter includes the received parts into the inventory by forwarding them to the base function (cf. `processDelivery`). Finally, `null` is returned because no further handling is envisaged, and the JDK's `Void` class is used as the result type to indicate this circumstance. The depicted behaviour is identical to the `deliver` method described above. The `processCallback` field can also be set via an anonymous inner class.

Ordering. The factory requests a set of parts by calling:

```
Active.promise(this, () -> supplier.order(partCount))
      .then(this.processCallback);
```

The semantics of this concatenated statement are noted below:

- `Active` is a library class providing utility functionality.
- Once again, `partCount` is a positive integer.

- The referenced supplier declares an active method with the signature:
  `Part[] order(Integer amount)`
- The `promise` operation instructs the given active invocation (cf. `order`) and passes back the associated promise.
- The previously created callback is installed by using the promise's `then` method.
- Thus, `processDelivery` stores the produced parts.

Advantages over futures. This sample illustrates the benefits of promises. Since the invocation of `order` is always one of the last statements, a future can only be resolved in two suboptimal ways:

- Blocked waiting prevents other calls from being answered. Hence, the remaining items cannot be distributed in the meantime.
- Delayed processing introduces further complexity and overhead through the necessary code. Because the delivery can only be incorporated when a method is performed, the downtime between various requests cannot be exploited.

As a consequence, promises have a significant advantage in this type of situation.

Comparison

Each of the two presented interaction patterns has its own merits.

- Explicit callbacks can be stored and invoked once the request is completed without committing to a specific moment. In particular, they can be executed even before the operation finishes if lengthy postprocessing slows down the termination.
- The implicit style enables natural method definitions since a procedure can simply return a result instead of sending a reply directly. Moreover, the handling operation can be declared `private` to reduce the attack surface, whereas an explicit callback can be shared and notified many times with malicious intent.

However, sometimes utilising the implicit technique is problematic. Section 3.5 already discussed aggregate values and nested data structures (cf. `Promise<Promise<Integer>>`). For the second challenge, the library offers a solution. The `promiseFlat` operation automatically unwraps the enclosed instance without blocking. This mechanism allows the caller to directly reference the outcome of the actual promise.

7.2 Lines of Code

The LOC metric estimates the effort of developing a specific program. However, the basic variant is of limited use in the context of this thesis. To collect additional information about the concepts, further specialisations concerning the synchronisation characteristics are gathered.

Focus. The assessment concentrates on interesting differences between the conventional and proposed approaches. The active object implementations are only briefly compared at the end of this section. They deviate in finer aspects from each other.

Measurements. The results are compiled into Table 7.1. Its columns have the following meaning:

- *Implementation* specifies the approach corresponding to the presented data.
- *Total LOC* covers the entire application. This number spans the basic features and also synchronisation, runner and utility classes. The code measuring the performance is excluded from this metric.
- *Synchronisation Class LOC* relates to the types coordinating the primary functionality. For this purpose, they were separated from the fundamental implementation of the system.
- *Synchronisation Statement LOC* counts the instructions controlling parallelism. This calculation includes the `ActiveObject` and `ActiveMethod` annotations as well as the built-in constructs. In the latter case, a fully `synchronized` method receives a value of one. In contrast, both the beginning and ending of `synchronized` blocks are taken into account because the closing bracket's positioning is a crucial detail regarding race conditions. Signalling invocations like `wait` and `notify` are factored in here as well.
- *Synchronisation Code LOC* describes the number of statements affected by concurrency control. This calculation comprises the corresponding methods, `synchronized` blocks and the code enclosed by these elements.

Implementation	**Total LOC**	**Synchro. Class LOC**	**Synchro. Statement LOC**	**Synchro. Code LOC**
Conventional	2077	459 (22.10%)	126 (1.06%)	94 (4.53%)
Active Objects (Explicit)	1839	234 (12.72%)	20 (0.69%)	70 (3.81%)
Active Objects (Promise)	1909	207 (10.84%)	20 (0.57%)	67 (3.51%)
Active Objects (Hybrid)	1844	220 (11.93%)	20 (0.65%)	67 (3.63%)

Table 7.1: Various lines of code (LOC) metrics of the implemented classes.

Total LOC.

- The conventional implementation showed the highest sum with 2.077.
- The active object concepts represented a reduction of 8.09% to 11.45%.

Generally, active objects are slightly favoured in this regard due to their simpler configuration. However, the gap is not always as large as indicated by this data. In our scenario, the ordinary components are wrapped by an extra layer that does not exist in the active object approaches. It stems from the detachment of the base functionality from the synchronisation code.

Synchronisation Class LOC.

- The traditional realisation showed a value of 459.
- The active object implementations required 49.02% to 54.91% less lines.

This variation is significant considering the former makes up 22.10% of the grand total compared to the 10.84%, 11.93% and 12.72% of the active object approaches.

Synchronisation Statement LOC.

- The conventional technique used 126 of these code lines.
- The other concepts managed coordination with only 15.87% of the above value.

The large deviation of the classic application is caused by the micromanagement of the synchronisation primitives.

Synchronisation Code LOC.

- The traditional program came last with 94 code lines.
- The other approaches utilised 25.53% to 28.72% less lines.

The substantially higher number of the ordinary variant is due to the manually implemented thread and the corresponding activation mechanisms (cf. `wait` and `notify`).

Implications. Although the difference of the total amount is not decisive in itself, the information gathered by the synchronisation metrics allows us to reinforce previous observations. As seen in Section 7.1, the usage of the concurrency control constructs is error-prone, and their handling is more complicated when compared to the library's features. Furthermore, the thread's `run` method and the inter-process communication introduce additional difficulties. Therefore, conventional components are more time-consuming to develop than active objects.

Active objects. The active object methodologies show roughly similar tendencies. Their results deviate marginally from each other because of the varying class layouts. As already mentioned, the explicit callbacks have an additional active method (cf. `deliver`) for receiving articles. Hence, it is subject to transformation. The implicit types do not provide these procedures and therefore came off marginally better in the LOC metrics.

7.3 Bytecode Size

The bytecode size determines the storage requirements of an application. Although the effects of this dimension are negligible in most circumstances, an imbalance of several orders of magnitude between the approaches cannot be tolerated.

Focus. The description concentrates on the instrumented classes. Their size illustrates the scale of the conducted modifications when contrasted with the base value. The increase must be considered if the adapted files are stored on disk for later use. The library provides an appropriate feature for this task.

Measurements. The information is presented in Table 7.2. The meanings of the individual columns are given below:

- *Implementation* states the approach relating to the figures.
- *Total Size* covers the fundamental components as well as the synchronisation, runner and utility classes. The system parts collecting the performance data are not included in the calculation.
- *Synchronisation Class Size* spans the types coordinating the core code. Therefore, the fundamental functionality was separated from the concurrency control mechanisms.
- *Total Instrumented Size* comprises the entire transformed bytecode, including all added and manipulated classes. For the latter, only the new value is used by the computation. System elements measuring the performance are not targeted by this criterion.
- *Instrumented Synchronisation Class Size* consists of the converted files belonging to the types that monitor the parallelism. They are the only ones treated by the agent. This category covers modifications of the servants and the creation of proxies.

Implementation	**Total Size (KB)**	**Synchro. Class Size (KB)**	**Total Instr. Size (KB)**	**Instr. Synchro. Class Size (KB)**
Conventional	124.35	23.46 (18.87%)	124.35	23.46 (18.87%)
Active Objects (Explicit)	112.72	12.33 (10.94%)	154.47	54.08 (35.01%)
Active Objects (Promise)	124.84	11.93 (9.56%)	165.63	52.73 (31.83%)
Active Objects (Hybrid)	116.90	11.74 (10.05%)	158.25	53.09 (33.55%)

Table 7.2: Comparison of the bytecode size of the different implementations in kilobytes.

Total Size.

- The explicit approach used the least storage with 112.72 kilobytes.
- The hybrid concept occupied 4.18 kilobytes (3.71%) more than the above.
- The conventional application needed further 11.63 kilobytes (10.32%).
- The promise implementation allocated additional 12.12 kilobytes (10.75%).

As mentioned in Section 7.1, the implicit callbacks are initialised through lambda expressions and anonymous inner classes. For each assignment of this kind, the compiler generates auxiliary files. This behaviour explains the ranking of the active object applications. The conventional program's higher requirements are due to its extra intermediate layer already mentioned in Section 7.2. Nevertheless, the collected information suggests the developed variants do not differ substantially in this matter.

Synchronisation Class Size.

- The active object applications used between 11.74 kilobytes to 12.33 kilobytes.
- The traditional program consumed 23.46 kilobytes.

Promise-based applications score slightly better in this category because the previously mentioned auxiliary files belong to the fundamental implementation. Therefore, this part of the bytecode is not included in the value.

Total Instrumented Size.

- The classical style was unaffected by the transformation and remained at 124.35 kilobytes.
- The active object realisations rose by between 40.79 (+32.67%) kilobytes and 41.75 kilobytes (+37.04%).

The size of the active objects grew quite noticeably. Hence, the traditional variant has a clear advantage in this regard. The applications relying on explicit callbacks exhibited a negligibly higher increase because of their additional active methods (cf. `deliver`).

Instrumented Synchronisation Class Size.

- Again, the conventional program stayed unaltered at 23.46 kilobytes.
- The active object applications were boosted by between 41.75 (+439%) and 41.35 kilobytes (+452%).

This massive change demonstrates the downside of implementing the active object pattern manually and brings the relief realised by the presented concepts to light.

Implications. The large extension of the file size shows the scale of the automated bytecode generation. This practice supports a small base disk space requirement comparable to the conventional approach. Even the total instrumented size is likely not an issue for the tasks to be solved by this methodology. Hence, the storage needed by the bytecode does not pose a problem when compared to the conventional style.

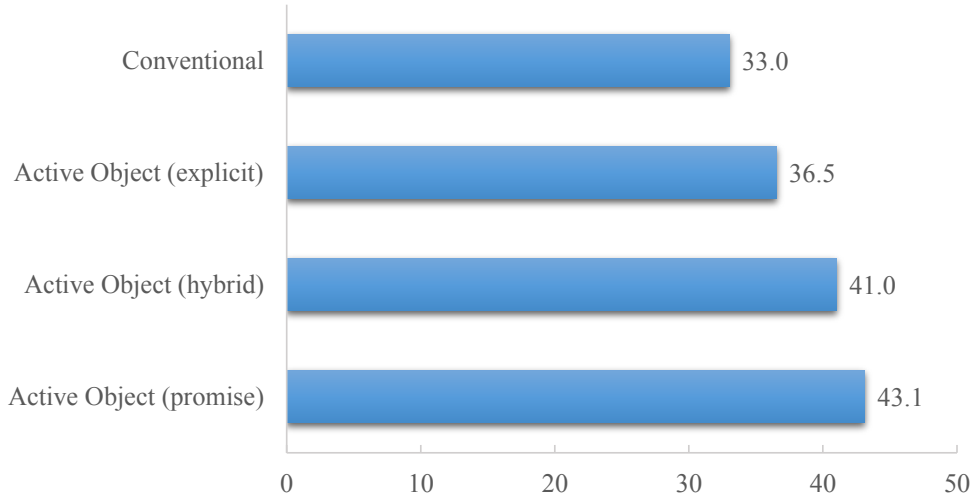

Figure 7.1: Timed duration of the individual implementations for dealing with 10,000 Customers. The number is given in seconds (lower is better).

7.4 Customer Timing

The ability to serve a sudden surge of requests is an interesting characteristic in the described scenario. Figure 7.1 visualises the durations needed by the developed solutions to satisfy ten thousand customers.

Result overview.

- The traditional approach was the fastest of all four taking 33 seconds.
- The explicit approach was 3.5 seconds (10.6%) slower.
- The hybrid approach was by 8 seconds (24.2%) behind.
- The implicit callback concept suffered a slowdown of 10.1 seconds (30.6%).
- If we only take the active object implementations into account, the slowest realisation took 6.6 seconds (18.2%) longer than the best one.

The two programs using the library's promise functionality came last in this category. They degraded the burst performance noticeably.

Implications. These values advocate using tailored interaction patterns in high-performance applications so as to take advantage of the speed of direct calls. Regarding this style, the classic realisation scores best. It employs built-in synchronisation constructs and avoids the slowdown of active invocations experienced by the explicit approach. Promises are less efficient because

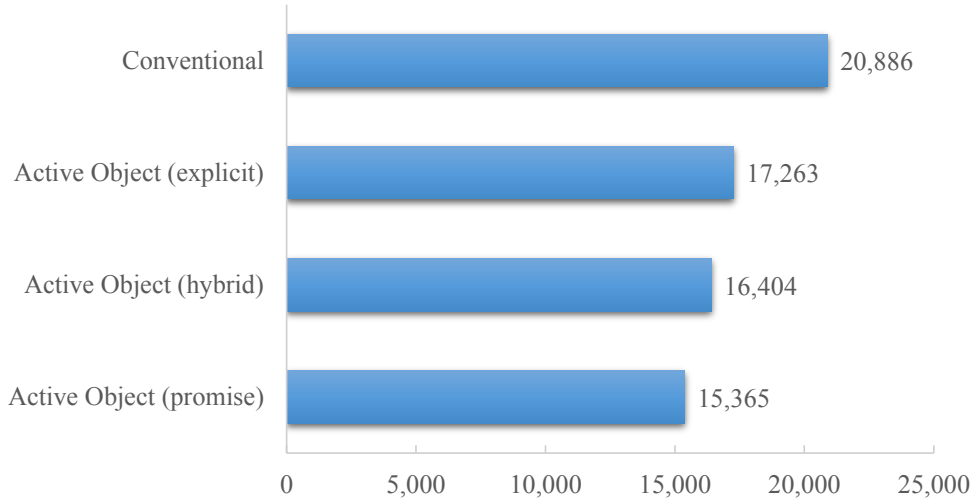

Figure 7.2: Number of customers finished by the different realisations when serving 1,000 simultaneous customers (higher is better).

they introduce overhead whenever an assigned callback is invoked. Hence, the hybrid is faster than the implicit concept since the former relies partially on direct communication. A sophisticated implementation of active objects and promises integrated at the language-level could eliminate these drawbacks.

7.5 Customer Throughput

This criterion assesses the capability of completing as many sales requests from a constant stream as possible. The results of this property are illustrated in Figure 7.2.

Result overview.

- The traditional approach performed best, finishing 20.886 clients per minute.
- The explicit design satisfied 17.3% less orders in the same period.
- The hybrid approach suffered a loss of 21.6% purchases.
- The implicit concept processed 26.4% fewer customers.

The traditional approach was ahead of the proposed competitors by a remarkable margin. Besides this facet, the active object applications only varied by 11%. This difference does not seem very large given the benefits of implicit callbacks.

Implications. This outcome is strongly orientated towards the result of the timing measurement. Hence, the statements made above do also apply to this category.

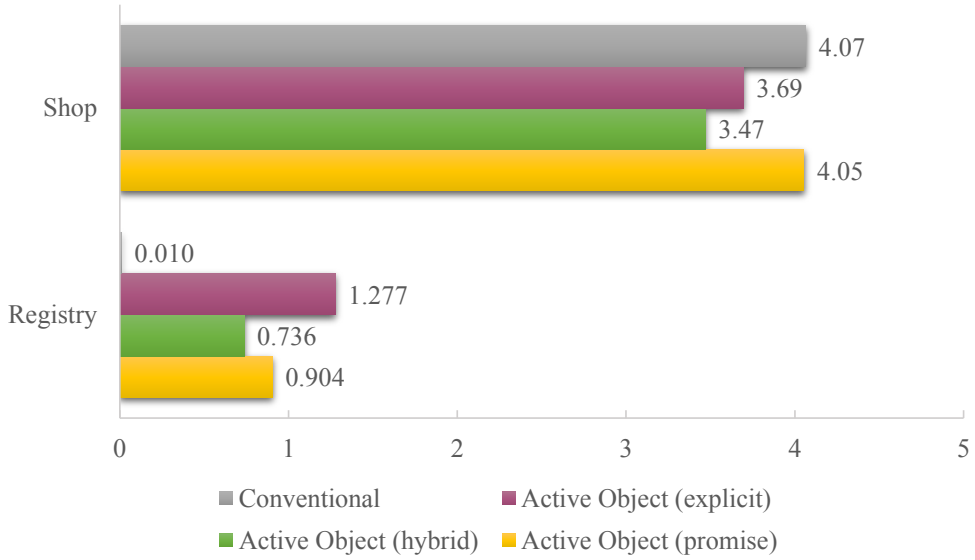

Figure 7.3: Average response time of the applications in milliseconds during the timed run (lower is better).

7.6 Component Response Time

A crucial characteristic of real-time applications is their responsiveness. It affects the way clients interact with the services and influences the user experience decisively. This section analyses the suitability of the active object concepts in this regard and compares them to the traditional approach.

Focus. In the presented scenario, `Customer` instances communicate synchronously with the `ShopRegistry` and `Shop` types. Thus, the evaluation concentrates on these components. The response times of the timing and throughput runs are depicted in figures 7.3 and 7.4 respectively.

Joint use. Some active object programs utilise the same component implementation. In these cases, fluctuations are caused by the interplay of the different implicit threads. The similarities are summarised below.

- All three techniques have a common `ShopRegistry` class.
- The hybrid and explicit practices share one `Shop` class.
- The promise system features its own `Shop` class.

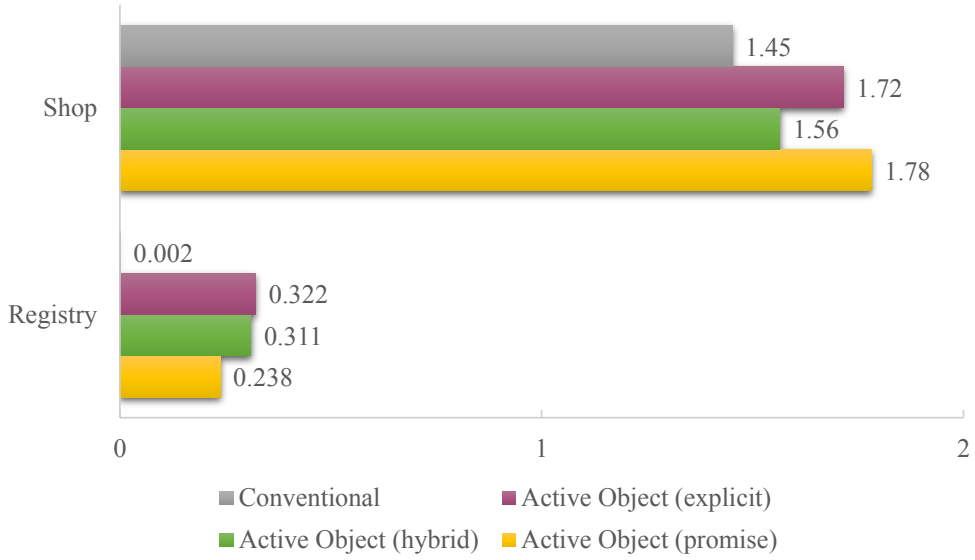

Figure 7.4: Average response time of the programs in milliseconds while performing the throughput test (lower is better).

Shop registry.

- *Throughput.* The standard methodology required a minuscule 0.002 milliseconds per request. The fastest active object approach was slower by a factor of 119.
- *Timing.* The ordinary application reacted after 0.010 milliseconds. The runner up needed 73.6 times this value.

Both results were overwhelmingly in favour of the traditional approach. The cause of this large gap must be sought in the simple class structure. A registry delegates each invocation to a held `Collection` instance. The overhead introduced by the active calls outweighs the runtime required by the swiftly executed core code. As a consequence, the active object's thread has to be restarted frequently. The speed of the built-in concurrency constructs further broadens this difference.

Shop.

- *Throughput.* The reaction time of the best active object approach was only 7.9% behind the conventional style.
- *Timing.* The traditional methodology was even slightly inferior to the least efficient active object approach, in concrete by 0.3%.

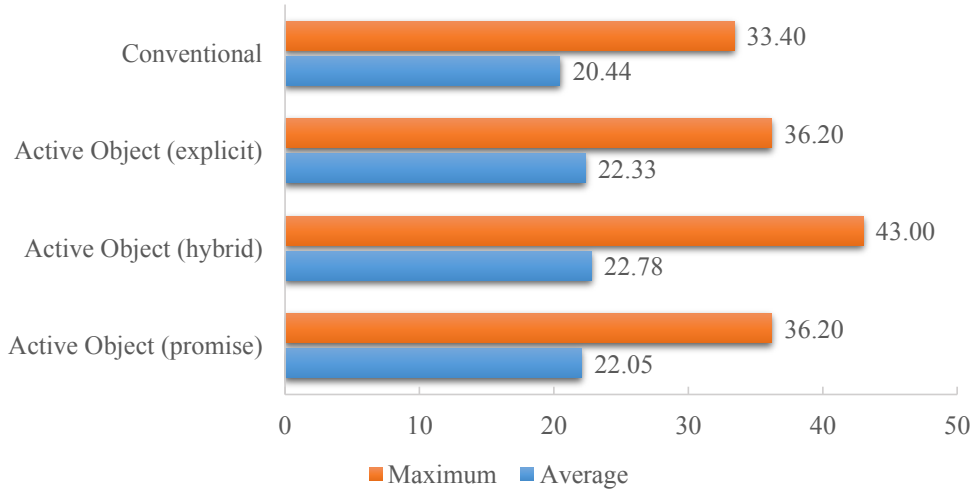

Figure 7.5: The average and maximum memory consumption of the approaches during the throughput run given in megabytes (lower is better).

This result reinforces the above explanation. The `Shop` class is more elaborate and features relatively expensive operations. Therefore, the cost of active invocations does not overshadow the complexity of the real procedure.

Active objects. If we constrain the analysis to the proposed concepts, both measurements draw a similar picture. The hybrid approach exhibited the best score, followed by the explicit callbacks and then the promises. Recall that the former two variants share the same `Shop` and `ShopRegistry` class. This circumstance explains their better response time. Hence, promises must be employed selectively in performance critical applications.

Implications. This outcome suggests plain classes should be realised conventionally. Usually, their implementation is rather simple even though the built-in concurrency constructs are used. The ordinary `ShopRegistry`, for instance, declares only fully `synchronized` methods. They delegate the call to the internal `Collection` object. On the contrary, sophisticated functionality can be programmed in either fashion because the performance is comparable. In some situations, the active object approach is actually faster responding than the traditional style (cf. `Shop` component).

7.7 Memory Consumption

In this part of the evaluation, we address the memory consumption of the variants. Figure 7.5 contrasts the measurements of the throughput run.

Average.

- The ordinary approach used the least memory with 20.44 megabytes.
- The active object based realisations used 1.61 (7.9%) to 2.34 (11.4%) megabytes more.

In this category, the difference was minor considering the worst and the best candidate are separated by less than 2.5 megabytes.

Maximum.

- The conventional approach had the lowest requirements using 33.4 megabytes at most.
- The explicit and implicit callback concepts both lost by 2.8 megabytes (8.4%).
- The hybrid application had a 9.6 megabytes (28.7%) higher demand.

The latter gap is dictated by two values in the raw data. The individual measurements are 36, 36, 37, 52 and 54 megabytes. The outliers are probably caused by the JVM's allocation strategy to avoid memory shortages. The other three contestants showed a more stable performance.

Implications. The traditional implementation is the slight favourite in both criteria. Furthermore, the hybrid approach is subject to variations in the chosen set-up. However, the disparity between the individual competitors is insignificant in most application areas, considering the memory technologies in this day and age. Note though that a minor difference can turn out to be crucial given a very high amount of concurrent components.

7.8 Final Thoughts

Throughout the evaluation, diverse arguments have been brought forward concerning the incorporation of active objects. The main benefits of using the library are listed next.

- The services facilitate the programming of concurrent scenarios.
- The necessary code is automatically generated.
- Annotations enable an easy configuration of the provided features.

In contrast, the most important drawbacks are summarised below.

- The performance is negatively affected.
- The memory requirements are increased.
- The instrumented class files consume more storage.

Hence, choosing between active objects and a custom solution depends on the particular use case. The following paragraphs give advice regarding specific situations.

Simple classes. The approach should generally not be applied to straightforward classes (cf. delegate pattern) because the introduced overhead outweighs the runtime of the operations by a large factor. If the instance's methods are rarely used or return very fast, a new thread needs to be started on every active invocation. In this circumstance, the response times are considerably lengthened and the performance is severely degraded.

Complex classes. The primary candidates for the proposed technique are classes wrapping sophisticated behaviour. As shown during the analysis, the runtime impact of the instrumented code is limited if applied to relatively long running methods. As an example, a few added milliseconds are irrelevant when downloading a file from a network node within several seconds.

Performance requirements. A concurrent system consists of multiple interacting elements. Typically, not all of them have to be highly-efficient in order to meet the specification. As a result, the presented performance characteristics are not always the deciding aspects for a project. For instance, envisage a background program that must process incoming orders and integrate them within two minutes. Hence, utilising active objects is certainly not ruled out from the start.

Bottlenecks. Extensive applications normally have a particular bottleneck with a critical impact on their performance. Such hot spots can be caused by database access, exceptional algorithms or network communication for example. In these cases, the small overhead introduced by the proposed realisation is negligible. Moreover, an active object can be exchanged easily if it becomes the limiting entity at some point. The new class can be tailored to special needs exposed during operation.

Storage. The bytecode size of the base program is comparable to the conventional approach. Furthermore, the size increase of the instrumented class files is only significant to the smallest spectrum of computing (cf. chipcards). These devices are not intended field of the library, because they do not have the resources to support adequate performance when used in highly-concurrent scenarios. Hence, such computing systems require custom behaviour.

Memory. The memory consumption is probably the most decisive reason against the use of the library's capabilities. The marginally raised requirements limit the number of concurrently active components. Imagine a web application instantiates a separate agent for each signed in client. Given the same hardware, the increased memory usage would imply an overall lower number of simultaneously logged in users. Depending on the project at hand, this drawback may prove critical.

Consequences. The discovered disadvantages are not determining arguments against the proposed solution in most use cases. A practical strategy is to develop the first version of the program by means of the provided constructs. Some classes can later be replaced if deemed

necessary. Optimising the library's core components and the generated code would further mitigate this problem. However, the speed of the built-in constructs is probably not attainable without native implementations.

CHAPTER 8

Conclusion

To conclude, let us set the thesis's problem statement indicated in Section 1.2 into the context of the findings. To this end, this chapter summarises the evaluation results and emphasises their implications. The last main segment describes possible future work.

Problem Statement. In the beginning, we posed two questions:

- Which functions and features are useful and sensible in an active object implementation to enable effective development of concurrent scenarios?
- How can the identified functions and features be realised in a library for Java by means of reflection and bytecode instrumentation?

The primary reasons for these points are the complexity and the potential defects caused by Java's built-in concurrency primitives. Solving rarely occurring deadlocks and race conditions usually involves high effort and may not be feasible altogether.

Findings. A model was developed having the above issues in mind. It expands on the idea of the basic active object pattern and incorporates advanced topics not considered by the related research. The design was then realised as a proof of concept to study the pros and cons of the approach by means of a concurrent scenario. The collected information showed that the active object applications' code was less cumbersome and involved a smaller number of potential synchronisation bugs in comparison to the traditional program. On the other hand, the analysis also revealed performance issues regarding the implementation. This circumstance was to be anticipated since the library was not tailored to be highly efficient. While the conventional concept certainly has a clear lead in some criteria, the difference is not as overwhelming as expected. As a consequence, the translation of the active object pattern into the proposed model proved to be adequate and sound. The automated instrumentation eases the application development and reduces the amount of boilerplate code effectively. In contrast, the advanced features appeared

to be more situational. Passive components primarily communicate via synchronous invocations, whereas active objects usually handle results using callbacks and placeholders to prevent temporal coupling. This practice avoids deadlocks and increases method request throughput. In this respect, promises represented a clear improvement over futures in particular circumstances while retaining exactly the same advantages. However, even this mechanism is unfit for some scenarios. Thus, they still must rely on explicit callbacks.

Implications. To summarise, the outcome of the evaluation implies that the automated active object approach is suitable for realising concurrent applications. Even though the advanced techniques are not always beneficial, they are reasonable enhancements of the model and have merits in a variety of situations. Optimisations and native implementations of the core components would help to resolve the remaining doubts.

Performance. Future work should address the aspects limiting the performance of the proposed model and the active object pattern as a whole. Mitigating these restrictions would expand the concept's competitiveness. This argument is substantiated by the growing popularity of Erlang's efficient actor implementation. For this purpose, the underlying weaknesses have to be examined thoroughly.

Extensions. Another interesting task is the study and realisation of additional extensions, and their introduction into the proposed design. This subject includes the actor model's location transparency and mobility properties. They were not taken into account in this thesis due to their scope. Multi-threading of active objects could also be incorporated. However, the chosen mechanism would have to be more refined than the one demonstrated in [22] and consider the criticism expressed in Chapter 2. Another notable topic with regard to the presented library is the possibility to use custom component classes, for example in place of `Scheduler` and `ActivationQueue`.

Interaction patterns. As seen during the code analysis, further research is necessary to completely replace explicit callbacks with an implicit technique. Promises already solve a part of the problem through the callback assignment operations. They should be complemented by an easily configurable feature that automatically correlates requests to delayed responses. Therefore, it has to separate the method's result from the actual return type and provide operations to access and set the underlying promise.

Outlook. In view of the projected increase of CPU cores in the next decades and progressively growing more complex applications, an adequate concurrent programming paradigm is needed to handle the myriad of spawned threads. As this thesis showed, the active object pattern is a suitable solution for such situations. Nevertheless, the concept requires more research to compete successfully with Java's standard concurrency constructs. This work contributed in part to this, but additional effort is still necessary to achieve the goal.

APPENDIX A

Concurrent Scenario

This appendix addresses the individual entities mentioned in Section 6.1 more closely. Figure A.1 gives an overview of their functionality and connections. The following descriptions also indicate parameters to control the execution.

A.1 `Part` and `Item`

`Item` instances represent the main products sought by customers and are manufactured by combining specific `Part` objects. The latter are the scenario's basic building blocks because their creation does not require any materials.

Manifestations. The `ItemType` and `PartType` enums list the various kinds of items and parts in the application. All `ItemType` constants have an associated construction plan stating the number and types of parts needed for the assembly of one instance.

Ordering information. `ItemOrder` is a utility class for acquiring items. It wraps a mapping of `ItemType` values to the desired quantities and provides methods to easily access these data. Conversely, parts are simply requested by specifying the necessary amount because each supplier only sells one part type.

Relationships. Multiple components reference and use the introduced entities, as is evident from Figure A.1:

- `Part` is used by `Supplier` and `Factory`.
- `Item` appears in `Factory`, `Warehouse`, `Shop` and `Customer`.
- `ItemOrder` is handled by `Factory`, `Warehouse` and `Shop`.

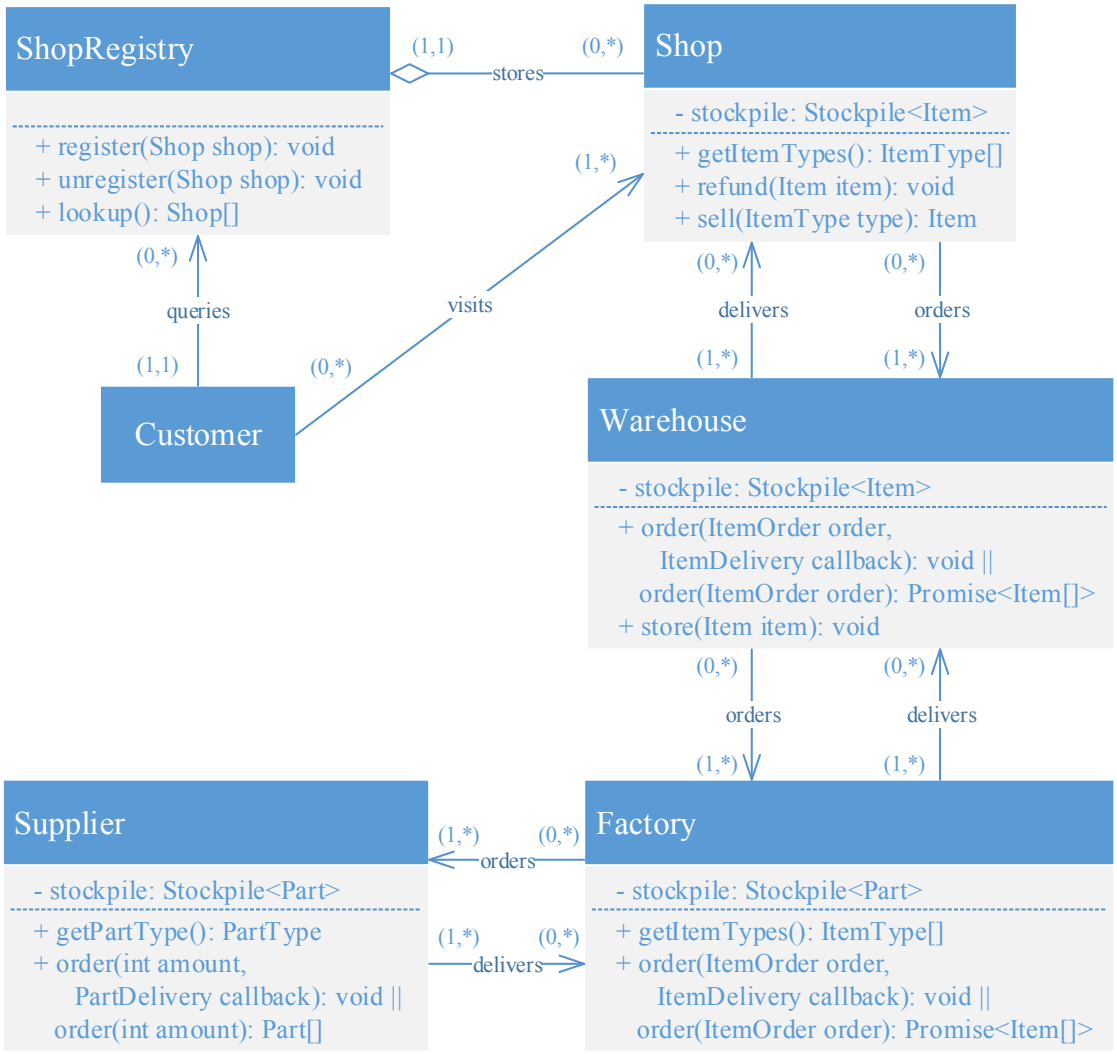

Figure A.1: The class diagram of the shop example.

Parameters. In the above context, the execution options relate to the number of `PartType` and `ItemType` values as well as the parts each item type consists of.

A.2 `PartDelivery` and `ItemDelivery`

In the explicit callback pattern, the `ItemDelivery` and `PartDelivery` interfaces decouple a distributor from the receiving object. The client has to implement one of them and pass itself as a parameter of the invocation to obtain the shipment. The two interfaces have a similar structure:

```
public interface ItemDelivery {
   public void deliver(Item[] items);
```

```
}
public interface PartDelivery {
   public void deliver(Part[] items);
}
```

The response is sent by invoking the corresponding `deliver` method with the requested products. Thus, the realised operation must process the incoming wares.

Implicit delivery. A class does not have to conform to this mechanism if the called procedure returns a promise or an array. An example of this situation is the use of the `Promise<Item[]> order(ItemOrder order)` operation offered by a `Factory` realisation.

A.3 `Stockpile`

The `Stockpile` instance is a vital element of most components. It provides features for storing articles and for querying information about the current inventory. However, the class itself is not thread-safe. Any access to such an object must therefore be controlled by the associated implementation.

Boundaries. A stockpile has two thresholds. Their meanings are described next:

- The *upper limit* states the maximum amount of products the stockpile can store per type.
- The *lower limit* specifies the minimum number of units that must be present of each kind.

The upper limit's effects are straightforward. If it is reached for a given type, a related object is simply discarded when transferred to the stockpile. The lower limit is a more interesting matter because it defines when the administered articles need to be replenished. In this situation, the stockpile's `hasNeededTypes()` function evaluates to `true`. The next paragraphs explain the impact of the boundary values in detail.

Reserve management. Every entity tries to maintain a basic set of articles to promptly deal with requests. In the best case, the stockpile holds as many instances of all kinds as its upper threshold supports. If the actual count of any type drops below the lower threshold, the inventory is completely restocked. Hence, the deviation between the current and highest number of every type is calculated and the difference is acquired. This task entails scheduling a new production run or triggers the placement of orders with the associated distributors.

Relationships. The `Stockpile` class maintains objects on behalf of `Factory`, `Supplier`, `Shop` and `Warehouse`.

Parameters. Both limits of a stockpile are independently customisable.

A.4 `Supplier`

A supplier contributes parts of one particular type to the manufacture of items. The generated `PartType` is configured through the constructor. This component features the members explained below:

- A `Stockpile` instance buffers pre-produced parts.
- `getPartType()` retrieves the defined, immutable `PartType`.
- The `order` methods request parts of the predetermined type.
- `order(int amount,PartDelivery callback)` invokes `callback` with the specified quantity.
- `order(int amount)` simply returns the stated number of parts.

Behaviour. A notable detail is that the `order` procedures always directly finish a received job. If the stock is entirely depleted while working on an shipment, the remaining parts are created immediately. If the stockpile's lower bound is undershot after a delivery, the supplier automatically starts a full production sequence.

Relationships. `Supplier` only communicates with `Factory`. However, the bond is decoupled by the functioning principle of `order`.

Parameters. This class features an interesting set of tuning options:

- The stockpile's upper and lower thresholds are adjustable.
- The manufacturing time of individual parts is modifiable.

A.5 `Factory`

A factory can build a particular set of item types out of parts. This information is passed to the constructor of the class. `Factory` implementations have the following elements:

- A `Stockpile` object stores the necessary parts to create the offered items.
- `getItemTypes()` obtains an array of the unchangeable types handled by an instance.
- The `order` operations are used by clients for acquiring items.
- `order(ItemOrder order, ItemDelivery callback)` performs `callback` when the shipment is ready.
- `order(ItemOrder order)` yields a promise which resolves to the desired items.

Erroneous calls. If an `ItemOrder` object contains an incorrect `ItemType` (cf. not indicated by `getItemTypes`), the `order` procedures throw an unchecked exception of the `IllegalArgumentException` class and ignore the invocation. This abnormal program behaviour must not occur during execution.

On-demand production. Items of a particular `ItemType` are only assembled when they are requested through an `ItemOrder`. Since a factory cannot anticipate incoming orders, it has an internal stockpile of parts to still be able to meet the needs in a short period.

Response. A factory may not hold all of the required parts when it receives an order. In this case, a delivery has to be processed before the associated invocation can be answered. For this reason, `order(ItemOrder order)` has to return a promise. The signature of `order(ItemOrder order,ItemDelivery callback)` remains unchanged, because `callback` can easily be invoked at any future moment.

Relationships. `Factory` utilises `Supplier` instances to obtain the necessary parts. At the same time, its functionality is only used by `Warehouse`, but this bond is decoupled by the employed reply mechanisms.

Parameters. The behaviour of the execution is determined through multiple settings:

- The stockpile's upper and lower limits are changeable.
- The assembly time of a single item can be customised.

A.6 `Warehouse`

A warehouse is an intermediate area for items created by factories and sold by shops. Such an instance is intended to carry every `ItemType`. The primary features of these entities are listed below:

- A `Stockpile` object is responsible for the retention of items.
- The `order` methods request items from the warehouse.
- `order(ItemOrder order,ItemDelivery callback)` sends the articles to the provided `callback`.
- `order(ItemOrder order)` returns a promise for accessing the inquired items.
- `store(Item item)` adds the supplied item to the internal stockpile if possible. The object is discarded when the maximum capacity of the given item's type is already present. The operation still succeeds in this case.

Response. When an `order` method is performed, some of the related items may not be available. Hence, `order(ItemOrder order)` fetches a promise instead of the actual set of items because a transmitted request cannot always be satisfied immediately. The signature of `order(ItemOrder order, ItemDelivery callback)` stays in line with the rest of the entities.

Relationships. This component maintains a direct connection to the `Factory` class to acquire the needed items. Moreover, the services of `Warehouse` are used by `Shop`. As with previous types, this relationship is decoupled.

Parameters. The only configuration options are the stockpile's upper and lower limit.

A.7 `Shop`

A shop is an element from which customers buy items from a certain range of types. During startup, every `Shop` instance announces itself to the main registry. Specific implementations include the characteristics listed below:

- A directly owned `Stockpile` reference stores the items.
- `getItemTypes()` queries the unalterable types an instance markets.
- `sell(ItemType type)` asks for an item of the given `ItemType`. If available, the procedure obtains an appropriate object. Otherwise, `null` is returned.
- If `sell` is called with an `ItemType` not carried by the shop, the invocation raises an unchecked `IllegalArgumentException`.
- `refund(Item item)` returns a bought item to the shop. The latter tries to incorporate the object into the stockpile. Although expected, the shop does not check whether it sold the supplied product.
- If the reserve of a refunded item's type is already full, the article is forwarded to a warehouse (cf. the `store` operation).

Relationships. `Shop` has strong ties to `Warehouse` and `ShopRegistry`. The `Customer` class is the sole client using the functionality of this component.

Parameters. The individual shops offer the possibility of adapting the limits of the held stockpile.

A.8 `ShopRegistry`

The `ShopRegistry` type gathers the active shops and provides this information to customers. An implementation must have the following members to enable dynamic coming and going of vendors:

- A `Collection` object contains the currently opened shops.
- `register(Shop shop)` enrols the given shop. A repeated addition is simply ignored. In this case, no exception is thrown.
- `unregister(Shop shop)` removes the supplied shop from the pool. Inapplicable deregistrations are ignored without raising an exception.
- `lookup()` yields an unsorted array enclosing the registry's latest shop roster. The function must not introduce an alias of the internal collection, but instead create a deep copy.

Relationships. Unsurprisingly, `ShopRegistry` has ties to the `Shop` type. Furthermore, this component is utilised by customers, but it does not hold any direct references to them.

Parameters. This type does not offer any customisations.

A.9 `Customer`

The `Customer` entities advance the application's execution state by buying items. Unlike the other classes, this one is shared between all approaches. A customer is represented by a standard Java `Thread` and receives a `ShopRegistry` instance at construction time. The associated `run()` method comprises one item sale and is structured as follows:

1. *Choosing a shop.* The customer invokes the provided registry's `lookup` function and retrieves the set of all active shops, one of which is chosen.
2. *Selecting an item type.* Next, the client queries the shop's `getItemTypes` method and picks one of the obtainable types.
3. *Purchasing an item.* The customer tries to buy an item of the determined `ItemType` by using the shop's `sell` operation.
4. *Out of stock.* If the previous invocation fails to acquire an object because the articles are unavailable, the process continues with Step 2.
5. *Refunding the item.* Once the demand has been met, the client can elect to return the item to the shop from which the article was bought.

As soon as the last step has finished, the `run()` procedure terminates and the customer is considered complete.

Randomisation. The decisions made in the process are based on pseudorandomly generated values. This practice aims to achieve a balanced distribution and introduces some unpredictable variations into the execution.

Relationships. `Customer` heavily depends on the services featured by the `ShopRegistry` and `Shop` classes.

Parameters. The percentage underlying the refund rate of items is globally adjustable.

Bibliography

[1] OpenJDK Repository: FutureTask. https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/concurrent/FutureTask.java (Archived by WebCite® http://www.webcitation.org/6SwwkZSZJ) [Online. Last accessed: 2014-09-29], 2011.

[2] Active Object Library JavaDoc. http://activeobj.bitbucket.org (PURL: http://www.purl.org/ph1l/mastersthesis/javadoc) [Online. Last accessed: 2014-09-29], 2014.

[3] Active Object Library Repository. https://bitbucket.org/activeobj/ajo (PURL: http://www.purl.org/ph1l/mastersthesis/repository) [Online. Last accessed: 2014-09-29], 2014.

[4] Polyglot - A Compiler Front End Framework for Building Java Language Extensions. http://www.cs.cornell.edu/projects/polyglot/ (Archived by WebCite® http://www.webcitation.org/6Ntj0VkE5) [Online. Last accessed: 2014-09-29], 2014.

[5] Gul A. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, 1985.

[6] Gul A. Agha and Carl Hewitt. Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism. In *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 19–41. Springer Berlin Heidelberg, 1985.

[7] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via Scheduling: Techniques for Efficiently Managing Shared State. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 640–652. ACM, 2011.

[8] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 2002.

[9] Joakim Carselind and Pascal Chatterjee. Concurrency on the JVM – An investigation of strategies for handling concurrency in Java, Clojure, and Groovy, 2012.

[10] Arunodaya Chatterjee. Futures: A Mechanism for Concurrency Among Objects. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, pages 562–567. ACM, 1989.

[11] Shigeru Chiba. Load-time structural reflection in Java. *ECOOP 2000—Object-Oriented Programming*, pages 313–336, 2000.

[12] Shigeru Chiba. Javassist. `http://www.csg.ci.i.u-tokyo.ac.jp/%7Echiba/javassist/` (Archived by WebCite® `http://www.webcitation.org/6Sewv2m3x`) [Online. Last accessed: 2014-09-29], 2014.

[13] Steven Clarke. Measuring API usability. *Doctor Dobb's Journal*, 29:S6–S9, 2004.

[14] Markus Dahm. Byte Code Engineering. In *JIT'99*, Informatik aktuell, pages 267–277. Springer Berlin Heidelberg, 1999.

[15] Frank S. de Boer, Dave Clarke, and Einar B. Johnsen. A Complete Guide to the Future. In *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer Berlin Heidelberg, 2007.

[16] John Fruehe. Planning Considerations for Multicore Processor Technology. *Dell Power Solutions*, pages 67–72. Dell, 2005.

[17] David Geer. Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13. IEEE, 2005.

[18] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice*. Addison-Wesley Longman, 2006.

[19] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification: Java SE 8 Edition. Technical report, 2014.

[20] Zahia Guessoum and Jean-Pierre Briot. From Active Objects to Autonomous Agents. *IEEE concurrency*, 7(3):68–76. IEEE, 1999.

[21] Ludovic Henrio, Fabrice Huet, and Zsolt István. A Language for Multi-Threaded Active Objects. Technical report, 2012.

[22] Ludovic Henrio, Fabrice Huet, and Zsolt István. Multi-Threaded Active Objects. In *Coordination Models and Languages*, volume 7890 of *Lecture Notes in Computer Science*, pages 90–104. Springer Berlin Heidelberg, 2013.

[23] Ludovic Henrio and Justine Rochas. Declarative Scheduling for Active Objects. In *SAC 2014 - 29th Symposium On Applied Computing*, SAC '14, pages 1–6. ACM, 2014.

[24] Christian Johansen. *Test-Driven JavaScript Development*. Addison-Wesley Professional, 1st edition, 2010.

[25] Einar B. Johnsen, Olaf Owe, and Ingrid C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66. Elsevier, 2006.

[26] Kennedy Kambona, Elisa G. Boix, and Wolfgang De Meuter. An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, DYLA '13, pages 3:1–3:9. ACM, 2013.

[27] Rajesh K. Karmani, Amin Shali, and Gul A. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, 2009.

[28] Dimitri Konstantas, Oscar M. Nierstrasz, and Michael Papathomas. An Implementation of Hybrid - A Concurrent, Object-Oriented Language. *Active Object Environments*, pages 61–105, 1988.

[29] Ralf Lämmel and Simon P. Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, pages 26–37. ACM, 2003.

[30] Ralf Lämmel and Simon P. Jones. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, volume 255 of *ICFP '04*, pages 244–255. ACM, 2004.

[31] R. Greg Lavender and Douglas C. Schmidt. Active Object: An Object Behavioral Pattern for Concurrent Programming. In *Pattern Languages of Program Design 2*, pages 483–499. Addison-Wesley Longman, 1996.

[32] Brad Long and Paul Strooper. A Classification of Concurrency Failures in Java Components. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 8–16. IEEE, 2003.

[33] Michael D. McCool. Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE*, 96(5):816–831. IEEE, 2008.

[34] Oscar M. Nierstrasz. Active Objects in Hybrid. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, volume 22 of *OOPSLA '87*, pages 243–253. ACM, 1987.

[35] Oscar M. Nierstrasz. Triggering Active Objects. *Objects and Things*, pages 43–78, 1987.

[36] Oscar M. Nierstrasz. A Tour of Hybrid: A Language for Programming with Active Objects. In *Advances in Object-Oriented Software Engineering*, Prentice Hall, pages 167–182. 1992.

[37] Oscar M. Nierstrasz. Regular Types for Active Objects. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 1–15. ACM, 1993.

[38] Oscar M. Nierstrasz and Michael Papathomas. Towards a Type Theory for Active Objects. In *Proceedings of the Workshop on Object-based Concurrent Programming*, OOPSLA/ECOOP '90, pages 89–93. ACM, 1991.

[39] Behrooz Nobakht, Frank S. de Boer, Mohammad M. Jaghoori, and Rudolf Schlatte. Programming and Deployment of Active Objects with Application-level Scheduling. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1883–1888. ACM, 2012.

[40] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin Heidelberg, 2003.

[41] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent Proxies for Java Futures. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 206–223. ACM, 2004.

[42] Martin P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *Software*, 26(6):27–34. IEEE, 2009.

[43] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732. Springer US, 2011.

[44] Jan Schäfer and Arnd Poetzsch-Heffter. CoBoxes: Unifying Active Objects and Structured Heaps. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS '08, pages 201–219. Springer Berlin Heidelberg, 2008.

[45] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 275–299. Springer Berlin Heidelberg, 2010.

[46] Jan Schäfer and Arnd Poetzsch-Heffter. Writing Concurrent Desktop Applications in an Actor-Based Programming Model. In *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, IWMSE '10, pages 2–9. ACM, 2010.

[47] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *Communications Magazine*, 37(4):54–63. IEEE, 1999.

[48] Savitha Srinivasan. Design Patterns in Object-Oriented Frameworks. *Computer*, 32(2):24–32. IEEE, 1999.

[49] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 104–128. Springer Berlin Heidelberg, 2008.

[50] Éric Tanter, Marc Ségura-Devillechaise, Jacques Noyé, and José Piquer. Altering Java Semantics via Bytecode Manipulation. In *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 283–298. Springer Berlin Heidelberg, 2002.

[51] András Vajda. *Programming Many-Core Chips*. Springer US, 1st edition, 2011.

[52] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. Advanced Runtime Adaptation for Java. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, pages 85–94. ACM, 2009.

[53] Steve Vinoski. Concurrency with Erlang. *Internet Computing*, 11(5):90–93. IEEE, 2007.