Technische Universität Wien

Masterthesis

Context-Aware Notification in Global Software Development

Institut für Softwaretechnik und interaktive Systeme
Technischen Universität Wien

unter der Anleitung von a.o. Univ. Prof. Dr. Stefan Biffl und Dr. Alexander Schatten

durch

Benedikt Eckhard
Mat.Nr.: 0204573

Oktober 2007

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzen Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Benedikt Eckhard
Wien, 15.10.2007

Abstract

Global software development (GSD) projects should allow efficient development of complex software systems as teams coming from diverse cultural backgrounds, technical capabilities, and time zones work together. However, GSD projects are also complex socio-technical systems with the challenge to collaborate in a heterogeneous and changing technical environment. Agility to react quickly and efficiently to changes is important, but hard to achieve with traditional process-driven methods and tools.

Project managers, group leaders, and architects of communication and coordination infrastructure in a GSD project need to provide a sufficient and minimal set of tools that support the timely exchange of relevant information for each role in the project.

This thesis presents:

1. A *model to describe GSD projects* as socio-technical systems and analyze their risks and weak spots in order to support the project manager in weighing the trade-offs of changes in the project plan between resources, quality and schedule.

2. A *method for notification modeling*: In a traditional software development project informal communication is important to identify challenges (typically linked to one or more events in the project) and discuss solution approaches. However, in GSD projects this communication is only available in local groups, but not necessarily between groups.

   An example is the need of several roles, which are not aware of each other, to coordinate their work. A common workaround is to notify many people of

important events via e-mail or shared lists. Unfortunately, this approach may lead to a role receiving too many irrelevant incoming messages or miss vital messages.

A good solution needs to build on an understanding of the project context (roles, technologies, processes, components, dependencies, etc.) and allows users to define precisely which changes in the project context should trigger notifications and who should receive them.

We present the *Notification Specification Language* (NSL) that allows a precise and user friendly modeling of the notification requirements.

3. The *tool prototype* NOTICON that can interpret the NSL and inform users according to the specifications timely and with minimal interruption of their primary work. The approach integrates via open source technologies (Mule, ActiveMQ, Drools) the work tools that are used in the GSD project in order to provide rich notification capabilities. Notifications are presented to the receivers in the user interface of the tools they are currently using; this minimizes interruption and additional tool overhead.

4. A feasibility study for the *evaluation of the model and the tool prototype* that was conducted in cooperation with experts from Siemens PSE in the context of a typical GSD environment. Based on this setting a "Total Cost of Ownership" analysis was done to compare NOTICON to a conventional "email per change" notification approach.

Important results of this work are: a) GSD projects face high communication risks and some of the risks can be mitigated with notification systems. b) Existing solutions are inapplicable because they generate either too much or too less notifications which causes high costs in GSD projects. c) the solution proposed in this thesis can be used to systematically describe and discover the notification requirements and can effectively and efficiently deliver 1) the right information, 2) at the right time, 3) to the right persons, without interrupting their current activities. This allows savings of more than 50% of the communication costs in big projects.

Danksagungen

Eine Diplomarbeit zu schreiben ähnelt einem klassischem Prozess in der Software-Entwicklung: Zuerst werden die Anforderungen erhoben, auf denen das Design der Arbeit erstellt wird. Dann erfolgt die Entwicklungs- bzw. Schreibarbeit und schließlich wird getestet und pünktlich aus- bzw. abgeliefert. Die Analogie zur Software-Entwicklung geht aber leider noch weiter. Tatsächlich ändern sich die Anforderungen regelmäßig und bei der Entwicklung stellt sich heraus, dass manche Dinge doch nicht so funktionieren können wie vorher überlegt und man wieder zurück zum Design muss. Damit dennoch eine gute Arbeit geschrieben werden kann, braucht es Personen, die einem in diesem Prozess ein bisschen zur Seite stehen. Ich hatte das Glück von vielen Personen unterstützt zu werden, bei denen ich mich herzlichst bedanken möchte:

Mein Betreuer Stefan Biffl hat mich manchmal zur Verzweiflung gebracht, aber viel öfter aufgebaut und motiviert und ist maßgeblich für die Qualität und Wisschenschaftlichkeit der Arbeit verantwortlich. Alexander Schatten hat mich als zweiter Betreuer besonders bei allen technischen Aspekten mit Ideen und Lösungsvorschlägen unterstützt. Matthias Meindl (Siemens AG) hat mit wertvollem Feedback zur Praxistauglichkeit der Lösung beigetragen. Dindin Wahyudin hat die Arbeit kritisch gelesen und viele gute Verbesserungsvorschläge gebracht. Michael Schmidt (uvd Business Consulting GmbH) hat die Arbeit großzügig gesponsert und spannenden Input aus der Praxis von GSD Projekten gegeben. Besonders bedanken möchte ich mich bei meinen Eltern, die mir das Studium überhaupt erst ermöglicht haben. Vielen Dank auch all meinen Freunden, die mich die letzten Monate kaum zu Gesicht bekommen haben, mich aber mit aufmunternden Mails versorgt haben. Zu guter letzt danke ich meiner Freundin Birgit Leidenfrost, die meinen ganzen Diplomarbeitsfrust abbekommen hat, mich aber dennoch fortwährend unterstützt und motiviert hat.

Contents

1 Introduction **1**

2 Related Work **7**

2.1 Global Software Development 7
2.1.1 Complexity aspects of GSD projects 10
2.1.2 Socio-technical Systems 14
2.1.3 Communication Risks in GSD Projects 16
2.2 Context-aware Systems 18
2.2.1 Introduction into context-aware systems 19
2.2.2 Common architecture of context-aware systems 20
2.2.3 Characteristics of context data 23
2.2.4 Context Modeling 26
2.3 Context-aware Notification Systems 27
2.3.1 Framework for Notification Systems 28
2.3.2 Overview on notification solutions 30
2.4 Event Stream Processing and Rule Engines 32
2.4.1 Event Stream Processing 34
2.4.2 Rule Engine 35
2.5 Enterprise Application Integration 37
2.5.1 Service Oriented Architecture 39
2.5.2 Enterprise Service Bus 41
2.5.3 Event Driven Architecture 43

3 Research Issues **45**

3.1 Research Questions 45
3.2 Research Design 46
3.3 Stakeholder Analysis 47
3.3.1 Stakeholder Classification Framework 47
3.3.2 Stakeholders 49
3.4 Requirements for the prototype 56
3.5 Notification Discovery and Description Framework and Process 58
3.5.1 Step 1: Analyze the project communication risks 60
3.5.2 Step 2: Describe the project organization 65

3.5.3 Step 3: Describe the key communications . . . 71
3.5.4 Step 4: Define the Notifications . . . 73

4 Prototype Development 77
4.1 Components . . . 77
4.2 Message Model . . . 81
4.3 Context Model . . . 83
4.3.1 Core Artifacts Layer . . . 84
4.3.2 Common Artifacts Layer . . . 88
4.3.3 User Artifact Layer . . . 89
4.4 Rule Engine Agenda . . . 89
4.5 Notification Specification Language . . . 90
4.5.1 Notation and General Concepts . . . 91
4.5.2 Title . . . 93
4.5.3 Receiver . . . 94
4.5.4 Context: . . . 96
4.5.5 Delivery options . . . 101
4.5.6 Channel specification . . . 102
4.5.7 Message content . . . 103

5 Case Study 105
5.1 Introduction . . . 106
5.2 Project Communication Risks . . . 106
5.3 Project Organization . . . 108
5.4 Key Communications . . . 111
5.5 Notification Definitions . . . 113
5.5.1 Failed test-case . . . 113
5.5.2 Daily summary of requirements that are ready to deploy . . . 114
5.5.3 Working on the same requirement . . . 115
5.6 User Interaction . . . 116

6 Discussion 121
6.1 Total Cost of Ownership . . . 121
6.1.1 Setting . . . 121
6.1.2 Model . . . 122
6.1.3 Conclusion . . . 131
6.2 Discussion of the Process . . . 132
6.3 Discussion of the Prototype . . . 134
6.4 Discussion of the Notification Specification Language . . . 136
6.5 Discussion of the Expected Benefits and Costs . . . 137

7 Summary & Future Work 141

7.1 Future Work 146
7.2 Takeaway Messages 147

1 Introduction

Lili is a developer in a big software development company that has offices all over the world. Currently she and about 70 other developers are working on an application consisting of several thousand components. This morning, Lili gets an Email from her chef who asks her to implement a new requirement concerning the calculation of interest rates. She reads the attached details and starts with the implementation. In the late afternoon on a coffee break, Lili meets her friend Kurt from the Quality Assurance team by chance. He tells her that they had to reset the test environment a few hours ago, because the customer changed its mind concerning some interest calculations. Lili worries that her work package is impacted too and calls her chef. She confirms that indeed the requirement Lili is working on had also changed. The hours that she already spent working on it were wasted time, but Lili is glad that she met Kurt - otherwise she might have worked for nothing for another day, as it had happened already in the past.

Situations like this are typical in software development projects and often caused by an inapplicable management of changes. In the past, software development processes, like the waterfall model, aimed to reduce the number of changes that occur during the project lifetime: First, the requirements are analyzed and written down in a contract that gets signed by both the customer and the software producer. Then follows a design phase where the whole system architecture gets defined which is implemented in the implementation phase. After that, the system gets tested and installed. Changes are only allowed within the phases and once a phase is completed its outputs must not be changed [20].

In practice this rigid process is not feasible in many projects because the general assumption that everything can be defined up front is often wrong [50]. Customers

want to change the requirements because of changes in the business environment, errors in the requirements elicitation phase, or just because of new ideas. Also the system architecture often needs to be adapted to better meet non-functional requirements like scalability and maintainability.

It was asked for more agility in software development processes - the ability to quickly react on changes, frequent delivery of business value, and more collaboration between developers and business experts [1]. Software development processes like *Extreme Programming*, *Scrum*, and the *Rational Unified Process* provide guidelines on how to efficiently develop software that builds on these premises. A key criteria for the success of agile projects is a frequent, effective, and efficient communication between the project members.

Global software development (GSD) projects should allow efficient development of complex software systems with teams coming from different regions of the world working together. However socio-technical aspects like different cultural backgrounds, different technical infrastructures, different languages, and different timezones, make communication between the project members much more difficult [39]. Typical problems are:

- A lack of awareness, so that project members do not know what is going on in the project and what the other teams are working on.
- Inefficient communication because of geographical distance, different time zones, and different languages.
- Incompatibilities between people, processes, and technology.

Thus, applying agile processes to GSD projects poses significant challenges for all project members. A sufficient coordination and communication infrastructure is required that supports the timely exchange of relevant information for each role in the project [85]. But infrastructure is not enough: the potential communication risks for a project need to be analyzed and understood before efficient countermeasures can be taken.

This thesis focuses on the efficient communication of project events (eg. changes)

to all concerned project members. An event can either be the modification of an artifact (eg. the change of a requirement), or the occurrence of a specific situation in the project (eg. two developers are concurrently modifying the same component). Currently two strategies are widely used for communicating project events: 1) The person who triggers the event informally notifies all project members that he or she thinks are interested or impacted by it (eg. via phone). 2) Project members individually subscribe to notifications that can be sent by tools like SVN whenever an event (eg. a check-in) occurs.

Both approaches have serious flaws in the context of GSD projects: The informal communication works very well in small teams, where every project member knows what the others are working on. In GSD projects this is only the case within teams but rarely between them; thus, the risk that important events are not, or very delayed, communicated to concerned people from other teams is very high.

Also the second approach is unfeasible in GSD projects. One issue is that people from different teams (maybe even different organization) might not have the rights to access the tools where the events occur. They might not even know which tools the other teams are using, nor if they provide subscription mechanisms. Another issue is that many of the occurring events are irrelevant for most project members in their current working context and just interrupt the primary activity. Traditional notification mechanisms (like sending emails to a mailing list in case of an event) can lead to an overload of information for users and can seriously slow down their work. With a bad ratio of unimportant to important information, also the risk that important information is overlooked increases [17].

We developed the **tool prototype** NOTICON [1] that allows project members to specify their notification requirements on a very fine granularity, taking their current working context into account. Eg. a developer can specify that she only wants to get notified of a requirement change, if she recently worked on that requirement. This maximizes the number of relevant notifications that are delivered to users and minimizes the number of irrelevant ones. NOTICON tightly integrates into the working environment and presents notifications in the tools that a user is currently

[1] NOTICON stands for "Notification In Context"

using; thus, the interruptions that are caused by the notifications are significantly reduced.

For maximum usability, we developed the **Notification Specification Language** (NSL) that allows a precise specification of notification requirements and can be easily written and understood by business users. It abstracts the information that concern a user (eg. requirements) from the tools where this information is maintained. This facilitates the reuse of the specifications across projects. NOTICON integrates into various applications (eg. Subversion) to sense project events and deliver the notifications according to the specifications.

As already mentioned, it is required to analyze the project communication risks before efficient risk mitigation measures can be taken. We introduce a **model to describe GSD projects as socio-technical systems** and analyze their weak spots in order to discover key communications that can be automated with notifications. Key communications are communications that have a high impact on the win conditions of the stakeholders if they do not occur. Our model provides concrete guidelines for project managers on how communication risks can be discovered and described.

For evaluation of the model and the prototype, we conducted a **feasibility study** in the context of a typical GSD environment in cooperation with experts from Siemens PSE[2]. The results were that the number of false positives (notifications that users get, but are not relevant for them) and false negatives (notifications that users do not get although they are relevant for them) could be drastically reduced compared to typical "email per event" communication. We also showed with a **total cost of ownership analysis** that the notification costs can be reduced by more then 50%.

In chapter 2 we will give a detailed introduction into the characteristics of GSD projects. We also give an overview of context-aware systems and the technologies that form the base for our implementation. The first part of the chapter is intended for readers who want to better understand the differences of GSD projects compared to small scale projects. The second part targets readers that are interested in the technical challenges and solution approaches to develop systems like NOTICON.

[2]PSE stands for "Programm- und Systementwicklung"

In chapter 3 our research questions are described. It also contains a detailed stakeholder analysis that elaborates on the requirements for a good notification solution. The second part of the chapter describes the notification discovery and description framework and process that can be used by project managers to discover and describe the notification needs.

Chapter 4 gives a detailed overview of the system architecture and describes the notification specification language in great detail. It targets developers who want to extend NOTICON and administrators who want to better understand how it integrates into their IT infrastructure.

In chapter 5 we show on a realistic case study how the process and the tool is used. We also show some screenshots that give an expression on how NOTICON integrates into the users working tools. The chapter will be especially interesting for project managers who want to evaluate NOTICON for use in their own projects and estimate the effort that it takes to get started.

Chapter 6 will finally discuss the benefits of NOTICON based on a total cost of ownership calculation. Also the good and weak points of the process, the prototype, and the notification specification language will be described. The cost analysis will be most interesting for project managers, while the technical discussion is more focused on researchers and developers who plan to extend the system.

Finally in chapter 7 we will summarize our findings and give a short roadmap on what is planned for the future.

2 Related Work

This chapter will give a detailed introduction into the characteristics of GSD projects and the concepts and technologies this work is related to. Section 2.1 frames the context of this thesis; we elaborate on GSD projects as complex socio-technical systems and the communication risks that arise. In section 2.2 it is described what context-aware systems are and the commonly used terminology. Section 2.3 distinguishes publish-subscribe notification systems from context-aware notification systems and compares existing solutions to our approach. Section 2.4 introduces rule engines as core technology for our solution and explains the rational why we preferred it over event stream processing. NOTICON has to integrate with various systems; therefore, an overview of enterprise application integration will be given in section 2.5.

2.1 Global Software Development

Global Software Development (GSD) projects are complex projects where multiple distributed teams and companies work together to deliver software systems to its clients. The tasks the teams have to perform, and the artifacts they produce, are highly dependent from each other; thus, a steady communication between the team members is required.

But the communication mechanisms in GSD projects are commonly disrupted by: [39]

1. A *less and less effective communication* because of differences in time, location, language, organization, culture, and technology.

2. A *lack of awareness*, meaning that project members do not know what their colleagues are working on and how to get in contact with them.

3. *Incompatibilities* between people, processes, and technology, which can lead to misunderstandings, confusion, and duplicated work.

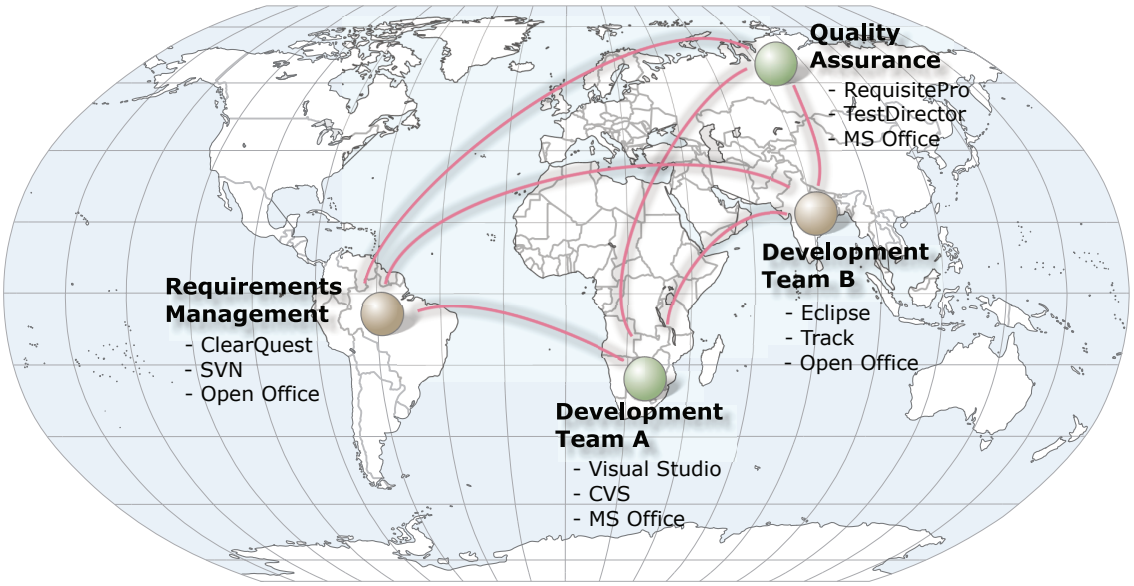

Figure 2.1: Overview of a GSD project

Figure 2.1 illustrates some of these issues with two companies (indicated by the different colors of the spheres) and four teams that are spread across the globe. The connections between the spheres indicate that communication is required between all teams, and the green boxes show that different tools fulfilling the same purpose are used in the project.

Tackling communication problems is difficult due to the complexity of the GSD projects that is caused by three main factors:

- The massive amount of heterogeneous elements like persons, source code, hardware, software, etc. that makes tool integration difficult.

- Many dependencies among the elements.

- A constantly changing environment.

A wide range of technologies (phone conferences, notification systems, wikis, email, instant messaging etc.) and processes (eg. stand up meetings) exist which try to enhance the communication between the project members. Notification systems are particularly suitable for automating key communication and increasing awareness. Key communication is communication that has a high impact when it does not occur, eg. if the change of an important component interface is not communicated, developers may have to rework a lot when the components are integrated later on.

We argue that the currently available notification systems are unsuitable for GSD projects, as they do not take enough the complexity aspects of these projects into account:

- They are often restricted to one platform or technology (eg. Windows, Java).
- They often cannot deal with heterogeneous elements to provide a uniform view of the project.
- They often do not provide means to reason over dependencies among elements.
- The means to define the notification requirements are too inflexible; thus, either too much or too less information is delivered to the users.

Our solution NOTICON has been designed particularly for GSD projects and aims to deliver the right information, at the right time, and at the right place. It can ensure communication where none would happen otherwise (because users might not be aware of dependencies) and automate key communication that is critical for the project, but easily overlooked or delayed eg. because of different timezones. NOTICON does not intend to replace other systems, but integrates with them to provide better usability and more relevant notifications.

Section 2.1.1 elaborates more on the characteristics that cause GSD projects to become complex systems. Understanding the different aspects of complexity is critical for a system design that is practical in real projects. Section 2.1.2 contains a description of socio-technical systems and communication as its central element. We set the focus on the social aspects and the theories about communication because these aspects are often overlooked in the design of technical systems. The findings

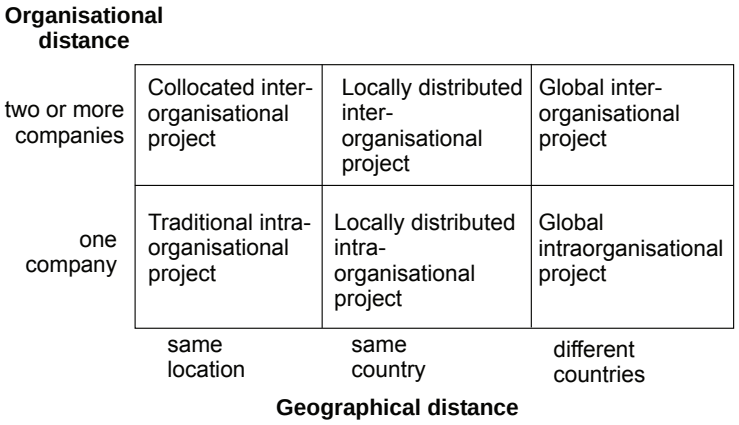

Figure 2.2: Project type classification [60]

serve as the theoretical background of the process we describe in section 3.5 on page 58. In section 2.1.3 the communication risks of GSD projects are described in detail.

2.1.1 Complexity aspects of GSD projects

The pressure to create software in less time, with higher quality, but cheaper, demands new methods to develop software [39]. Many organizations began to experiment with remotely located software development facilities and outsourcing, to better access skilled people and decrease the costs. Software became vital for almost every business, but its complexity increased.

The traditional intra-organisational project (figure 2.2) where all project work is done within one company at one location became unusual for bigger projects. Big projects are often developed by many companies spread over different countries. The solution we propose targets projects in the upper right corner of figure 2.2.

Tackling communication issues requires a detailed understanding of the complexity of GSD projects that is caused by three main factors: (1) many heterogeneous elements, (2) dependencies among the elements, and (3) a constantly changing environment.

Many heterogeneous elements

In GSD projects often hundreds of persons are involved in developing the system. They come from different countries, speak different languages, have different skills, attitudes, believes, and behaviors. The persons interact through tools (like OpenOffice or Eclipse) with elements like source code, requirements, or documentation.

NOTICON integrates data from various systems to provide a uniform view of all project elements. Eg. for a user concerned about persons and their relations, it is transparent that the data may come from different sources (eg. an Active Directory and a LDAP repository). Differing from other notification solutions, NOTICON displays its information in the tools that are already used in the project; thus, very flexible adaption and transformation mechanisms of information are required.

In co-located projects the infrastructure (hardware, software) is usually the same for all project members which makes application integration relatively easy. In GSD projects on the other hand the infrastructure can be different from location to location; thus, integration gets more difficult and flexible routing and transformation mechanisms are required.

Example: the development team in India uses Edgewall Track [1] to track the requirements, Eclipse [2] for development and Subversion [3] as code repository. The requirements management team in Brazil uses IBM Rational ClearQuest [4] for requirements management and the Quality Assurance in Russia uses Mecury Test Director [5] to run the tests and Subversion to store the component versions. Users may want to specify notification that reference "Source Code", "Requirement", "Testcase", and "Test Result" artifacts, regardless of the different tools that manage the data.

[1] http://track.edgewall.org
[2] http://www.eclipse.org
[3] http://www.subversion.org
[4] http://www-306.ibm.com/software/awdtools/clearquest/
[5] http://www.mercury.com/us/products/quality-center/testdirector/

Dependencies among the elements

Decomposing complex systems into smaller parts is a central principle of software architecture [3] and the task of project management. Different teams can then work relatively independently on their parts which are integrated with the other parts at some point. This would not pose a significant challenge if the parts where not highly dependent from each other and changes (eg. of requirements) were not the norm. Research shows (eg. [65, 16]) that dependencies create the need for communication between persons.

"A dependency between software modules is said to exist when a module relies on another to perform its operation or when changes to the latter must be reflected on the former." [25] The definition of Fonseca et al. is also valid for other elements in a software development project like activities, tasks, requirements, or tests. Management has to ensure that changes are communicated in time to all affected persons and its impact is held small.

Decoupling elements from each other, by using various software development techniques like Mock Objects, can reduce the immediate impact of changes; failures in one module do not hinder the development of dependent modules. But these techniques can lead to situations were different teams are not aware of the dependencies from each other, and failures are only discovered when the parts are integrated. Rigid specifications reduce the integration risks, but also the projects agility. Continuous integration and automatic test suites are another approach but often infeasible in GSD projects.

In GSD projects the number of dependencies is very high, and different roles and individuals have a different knowledge about them. Individuals are often neither aware of all dependencies of their work packages, nor the persons they could ask; thus, estimating the impact of changes is difficult and inefficient.

With NOTICON the knowledge of dependencies between elements can be made explicit and leveraged by all project members. Notification rules can use the dependency information to eg. make users aware of other developers that work on related artifacts, or inform them if the requirement that a component implements

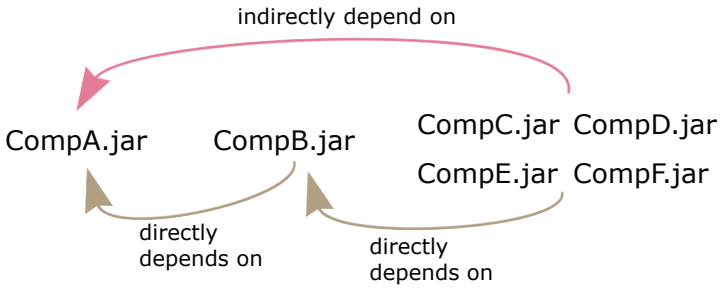

Figure 2.3: Unawareness of indirect dependencies can lead to wrong assumptions on the impact of a change.

has changed.

It is also possible to reason over all project elements to discover dependencies that where not known before as illustrated in figure 2.3: The components $C \ldots F$ are dependent from component B which is dependent from component A thus the components $C \ldots F$ are indirectly dependent on component A. A developer that just knows about the dependencies between B and A is likely to underestimate the impact of a change of A. Making these dependencies explicit allows users to make better decisions.

Constantly changing environment

Due to the size and length of GSD projects the project ecosystem [59] changes continuously. Persons come and go, software components are produced and modified, and requirements change during the whole project lifetime. Due to the dependencies between elements the changes need to be communicated effectively, but people are often not aware of each other; thus, communication is impaired or does not happen at all.

The goal of any notification system is to communicate these changes effectively to interested parties. In GSD projects changes occur much more frequently; thus, traditional approaches, where any change triggers a notification for all people that might be concerned, would lead to far too much information for an individual. NOTICON provides the ability to specify rules, that take a users current work context

into account which reduces the amount of notifications that will be transmitted, but increase their importance.

Example: in a small project developers might subscribe to a SVN repository to be notified whenever a commit occurs. This works well for small teams and results in a couple of mails per day. In projects with hundreds of developers, thousands of emails would be delivered each day, making the system useless. With NOTICON the information of a commit could be delayed until the user works with an artifact that is affected by it.

The constant changes also motivate the need to integrate applications. Adding another tool, where information about users, roles, requirements, etc. needed to be maintained, would introduce new possibilities for inconsistencies and the costs would exceed its value.

NOTICON adapts very well to a constantly changing environment because it requires only minimal manual maintenance. The configuration of NOTICON has to be done only once, and data about users, roles, security settings, etc. can be integrated from different systems. Changes (eg. a new user is added to a LDAP repository) can automatically be reflected in NOTICON.

In the last sections we described the complexity aspects of GSD projects from a technical viewpoint. The socio-technical aspects of GSD projects will be described next.

2.1.2 Socio-technical Systems

GSD projects are complex socio-technical systems where there is a steady interaction (1) between people, (2) people and technology, and (3) technology [77]. The characteristics of the systems have to be known to be able to analyze communication weaknesses and introduce tools that efficiently tackle them.

A system is defined in system theory as a web of relationships between entities that are differentiated from other entities within an environment. In a process called

autopoiesis a system continuously reproduces itself to sustain the differentiation [54]. Eg. an organization is a system within a legal environment that differentiates itself from other organizations by eg. offering different products and services. A social system, like a small group of developers, differentiates itself from other groups eg. by using a certain vocabulary or by sharing information that only the members of the group know. Social systems can only operate and reproduce themselves through communication, which is a three step process between Alter (the person who acts through communication) and Ego (the receiver) [10, 54]:

1. *Selection of the information*: Different persons have a different perception of their environment and consider different aspects of it as information that is worth communicating. Eg. a system administrator who starts an anti virus program might not consider this as important information. The first step for Ego is to select one out of the set of information he or she wants to communicate.

2. *Selection of the form*: In the second step, Ego translates the information into a message and selects the medium to communicate it (eg. by email). The content of the message is influenced by psychological, social, and cultural aspects as well as the expected reaction and knowledge of Ego [44]. Eg. if Alter selects "The requirement has changed" as message, it is assumed that Ego knows which requirement is meant. The selection also alters the search space for selective connectivity - the set of possible responses.

3. *Selection of an understanding*: In the last step, the receiving party has to interpret the message and transform it back to information. One message can be understood in different ways, and Ego selects one of the possibilities. Eg. "The requirement has changed" could be interpreted as "Requirement 2432 has changed" if previous communication was about requirement 2432.

The conceptualization of system and communication is very important for this work. GSD projects incorporate a wide variety of systems that share a common goal, but are nevertheless distinct systems which have to reconstitute themselves all the time. The characteristics of the systems influence all steps in the communication process. In particular the "selection of an understanding" step is prone to errors if no shared

vocabulary has been defined, or processes differ from system to system. Eg. the "Requirement X has changed" message could trigger an immediate reaction of the persons in system A but be perceived as "nice to know" by the persons in system B.

Socio-technical systems add another layer of complexity. They do not only include communication between people, but also interaction between people and technology. Sena and Shani [77] decompose a socio-technical system into three sub systems:

- The *social subsystem* with people, their knowledge, attitudes, skills and competencies.
- The *technical subsystem* as a system that converts input into some output.
- The *environmental subsystem* which includes competitors, customers, law, etc.

There is a complex interaction between the subsystems, but as Whithworth notes not much attention has been paid yet to this fact. He argues that "system designer must recognize accepted social concepts, like freedom, privacy and democracy, that is, specify social requirements as they do technical ones." [88].

This motivates the need for a model that is used to systematically describe and discover notification requirements, taking socio-technical aspects into account.

2.1.3 Communication Risks in GSD Projects

Herbsleb states that "the key phenomenon of GSD is coordination over distance" [39]. He defines coordination as "managing dependencies among tasks" and points out that without inter-location and inter-organization dependencies, GSD projects would not pose significant challenges. In smaller project, which don't span organizations or locations, project members have a shared view of how the work will proceed. They share a vocabulary and know how expertise and responsibilities are distributed. They are also aware of what their colleagues are doing and know how their work will affect the others. In contrary GSD projects have the following main problems [39]:

- *Less communication and less effective communication*: In GSD projects, people communicate with less people, less frequently, and the communication is less effective. The lack of effectiveness is often caused by delays in asynchronous communication. NOTICON can increase the amount of effective communication by automating some key communication. It ensures an effective delivery to all interested parties at the moment the information becomes relevant for them.

- *Lack of awareness*: "Awareness involves knowing who is 'around', what activities are occurring, who is talking with whom; it provides a view of one another in the daily work environments." [19]. According to Herbsleb [39] the lack of awareness makes it difficult to get in contact with other developers and leads to misunderstandings of communication contents and motivations. Most important is the fact that "it hinders a project's ability to keep track of the effects of change as they propagate across sites.". NOTICON is particularly suitable for notifications that increase the awareness because it can sense the current users context: eg. a notification could be sent to two developers when they are concurrently modifying the same file; or a notification could be delivered whenever a users does a checkout from the SVN repository, showing a summary of all important project activities since the last checkout.

- *Incompatibilities*: In GSD projects people from different cultures, with different experiences, different communication habits, and a different work environment work together. The incompatibilities across tools may hinder communication and demand error prune workarounds (eg. Meeting invitations that are sent from the MS Outlook calendar can hardly be read in Lotus Notes). They also restrict the ability to apply common processes across sites which can lead to misunderstandings and confusion. NOTICON can tackle some incompatibilities by adapting a notifications content to the users context. Eg. the notification can be localized based on the users preferred language.

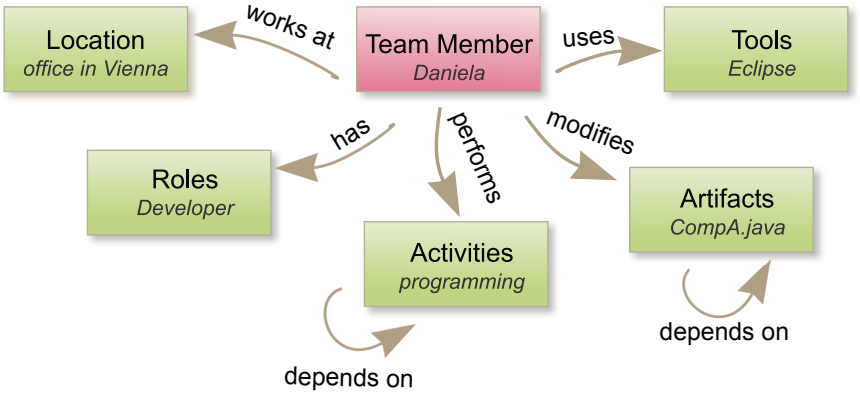

Figure 2.4: Example user context for notifications in GSD projects

2.2 Context-aware Systems

Context determines the usefulness of information for the receiver. Providing users with appropriate contextual information, allows them to better coordinate their work and make better decisions [72].

> "Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themself. [...] A system is context aware if it uses context to provide relevant information and/or services to the users, where relevancy depends on the users's task." [7]

Figure 2.4 illustrates a team members' context in GSD projects. He or she works at a specific location and performs some activities. Project members can have several roles and use tools like Eclipse to modify artifacts like source code. Artifacts and activities have dependencies as explained in section 2.1.1. The context changes frequently and the changes can trigger notifications.

NOTICON is a context-aware system because it adapts its behavior in three ways to the current user context:

1. NOTICON is able to automatically filter notifications that are currently relevant

for the user by analyzing the dependencies of activities and artifacts. This reduces the amount of notifications a user receives, but increases its relevance. It also reduces the need to manually subscribe to events that might be of interest.

2. The notifications are displayed in the tool that is currently focused. This increases awareness without interrupting the users primary task.

3. The content of the notification and its attributes (eg. priority) are adapted based on the projects context. Eg. the content of a warning notification could contain the contact information of the persons that are currently working on artifacts that might be influenced by the problem.

Section 2.2.1 gives a short introduction to context-aware systems and highlights the domains where these context-aware systems are used successfully. The general architecture of context aware systems and existing solutions are described in section 2.2.2. In section 2.2.3 we elaborate more on the different types of context data. Understanding its characteristics is important for the systems design.

2.2.1 Introduction into context-aware systems

People are very good at performing several tasks in parallel by concentrating on a single task at a given point in time. Different tasks require different information, thus a person's information needs are highly dependent on what he or she is currently doing. Too much or the wrong information would hinder the current activity. Application developers need to know what the users do and might want to know at the moment, and change the applications behavior accordingly.

A simple example is the context menu that appears when one right-clicks on an item in many operating systems. The tasks a user can perform from the menu are different depending on the clicked item and other context information (eg. if the application is currently performing some other tasks).

An example of an unsuccessful implementation is the interactive help in former ver-

sions of Microsoft Word: A little figure (bracket, puppy, etc.) popped up whenever you wrote a few letters and asked to help with either more information or by performing some little tasks. Unfortunately, the suggested tasks were most often useless and doing them manually was much quicker and accurate. The pop-up interrupted the users from their current activities and slowed them down. [24]

"The most profound technologies are those that disappear." [86]

According to Mark Weiser, most current technology demands too much attention from users and the better the technology gets, the less visible they become. He brought a simple pen as an example: we use it without even noticing and don't care how it works as long as it does. Following the track of Weiser the discipline of *ubiquitous computing* emerged: complex technology and computing devices should be embedded into daily goods. A music player could be embedded into a jacket, a credit card into a wallet, and so on. To really "disappear", devices should minimize user interaction and blend into the users environment. The credit card should automatically pay the bill in the restaurant and the music player should adjust its volume depending on the surrounding noise. To provide these capabilities, devices have to communicate with each other and become *context aware.*

The design of NOTICON was strongly influenced by the premise of Weiser. By integrating our tool into applications that are used anyway within the project, the notification system kind of disappears, and we hope that users just think of it as a separate tool when it is missing.

2.2.2 Common architecture of context-aware systems

Baldauf et al. [9] surveyed eight context-aware frameworks and deduced the common layered architecture [27] shown in figure 2.5. Their conceptual architecture also reflects the architecture of NOTICON as described in chapter 4.

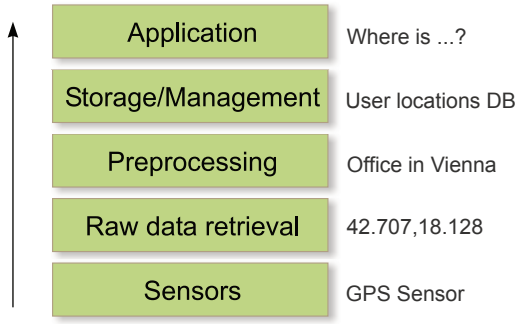

Figure 2.5: Architecture layers for context-aware systems [9]

Sensors

Sensors are responsible for monitoring the environment they are embedded in and offering this data to upper layers. Three different types of sensors can be distinguished:

- A *Physical Sensor* is a device that is able to capture physical data, like location (GPS), audio (microphone), or temperature (thermometer). Existing solutions are very much focused on this type of sensors, but for NOTICON they play only a minor role.
- *Virtual Sensors* obtain context data from software applications or services; eg. an Eclipse or MS Word plug-in that senses the current open file. Virtual sensors can be viewed as autonomous agents and can also perform complex activities like database lookups.
- *Logical Sensors* use a couple of information sources and enhance physical and virtual sensor data with additional information. A logical sensor could use the virtual Eclipse sensor described above, interprets its readings and guesses the activity that the user is currently performing.

Raw data retrieval

This layer makes the low level details of hardware access transparent. Eg. the GPS position must be accessed differently from device to device. The raw data retrieval shields application developers from these details and provides them with abstract methods like "readGPSPosition()" that return the data in a standard format eg. "42.707,18.128".

In NOTICON the ability to flexibly transform message into different formats provides a similar functionality. Eg. Rational ClearQuest and Edgewall Track are both able to send email messages when a requirement changes. Transformation rules can be specified that transform the different email messages to a common format that can be processed by other components.

Preprocessing

The preprocessing acts on the raw data and combines it to high-level information; this process is called "aggregation" or "composition". Also filtering mechanisms are installed at this layer to reduce the amount of data that is passed to the other layers.

The preprocessing layer in NOTICON is split into two sub-layers for performance and scalability reasons. The *context preparation* and the *context reasoning* layers:

1. The context preparation layer reduces the data that should be stored in the context model. Components like a correlated events processor [74] or an event stream engines like Esper can reason over the raw data and aggregate them to high level information that is then sent to the context reasoning layer.

2. The context reasoning layer is responsible for creating notification messages based on the changes in the context model. We use a reactive forward chaining rule engine that reasons over the data provided by the context preprocessing layer (see section 2.4).

Storage and Management

This layer provides access to context data and results of the reasoning (notifications) to applications and other components. The data is published in either push or pull mode:

- In *push mode* clients subscribe to data that they are interested in, and the system actively publishes new data to interested clients.
- In *pull mode* clients query the available data and thus can also get access to historical context data.

In NOTICON currently only the push mode is implemented; new notifications are automatically pushed to client components which display them to the users.

Application Layer

The application layer subsumes all clients that react on context events or make the data available to end users. Although NOTICON is designed to prepare information to be consumed by humans, nothing prevents different use cases. Eg. an application could automatically run integration tests if it receives a notification that two dependent components have been changed.

2.2.3 Characteristics of context data

We characterize context data by (1) the way it gets produced which can be either statically or dynamically, (2) its persistence requirements, (3) the level of imperfection.

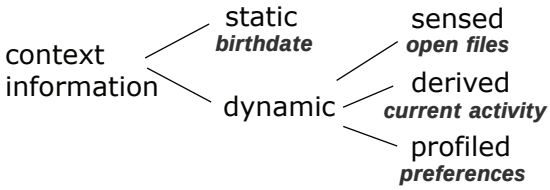

Figure 2.6: Context classification for the validity of data

Static or Dynamic

Henricksen et al. [38] characterize context information as either static or dynamic (see figure 2.6). *Static information* is something invariant, like a person's date of birth. *Dynamic information* is everything that changes more or less frequently, like an activity a person is currently doing, a piece of source code, or a requirement. Dynamic information is further classified into sensed, derived and profiled information.

- *Sensed information* is produced at the sensor layer described in section 2.2.2. The high frequency of which new data becomes available is characteristic for this type of information.

- *Derived information* is any information that is produced by aggregating and correlating sensed information. Eg. the activity of a person could be derived from the files that a person currently works on, and the used applications.

- *Profiled information* is information that cannot be sensed or derived and has to be provided directly by the user. Profiled information can be used to tailor the system to the specific needs of a user. Eg. he or she could state whether the notification was detailed enough, or more information should be included next time.

In the context reasoning layer of NOTICON we currently do not distinguish between the different types of context information. However the distinction is helpful for the analysis of which information can be sensed, which can only be derived, and which has to be provided by users. It also influences the persistence requirements and has different characteristics regarding its imperfection.

Persistence Requirements

The dynamic nature of context information and the different intervals at which data is gathered from the sensors require different persistence strategies [38]. Some sensor data might not need to be persisted at all, while a complete history has to be maintained for others.

Zimmer argues that "context history is most relevant in designing context-aware applications for it can influence the meaning of a certain context" [90]. Eg. the failure of a test case is something which might not be of much interest in GSD projects. If the test case continues to fail for a month, it might indicate a bigger problem.

System attributes like performance, maintenance, and reliability need to be balanced. Persisting more data allows more complex rules for reasoning and increases the systems reliability. But it has a negative impact on the performance of both the running system and the time it takes to recover from a system failure.

In NOTICON flexible routing rules define which type of information should be stored and which data should be recovered in case of a failure.

Level of Imperfection

Another property of context information is its imperfection [73]. Sensor readings can be incorrect, ambiguous or delayed [38] and a context aware application should be able to deal with these uncertainties. Information from logical sensors and derived information are particularly prone to these kind of errors. Several researchers suggested to assign quality metadata such as certainty and use this information in rules [83, 37].

Mechanisms for dealing with uncertainty are not implemented in the current NOTICON prototype and might be addressed in future work. Most of the data that is gathered from virtual sensors which monitor applications; thus, we expect a very low rate rate of incorrect sensor readings.

2.2.4 Context Modeling

A context model defines and describes the context data and the possible relations between the elements. Various techniques, ranging from simple key-value representation to ontology based approaches, exist and have been used successfully in context-aware systems. Strang and Linnhoff-Popien [81] defines six demands on a context modeling approach:

1. It should be easy to create context information from different devices.
2. It should be possible to validate the context information against the context model.
3. It should be possible to add additional characteristics to context data like quality and richness indications.
4. It should be possible to deal with ambiguity and incompleteness.
5. It should be applicable within the existing infrastructure.

In our prototype we use a Java class based context model. Context types (eg. "Requirement") are represented as classes and context data (eg. "Requirement 132") as instances of this class. Relations between objects are represented as separate objects. The model is described in detail in chapter 4.

NOTICON does not require sensors to use a particular context format - an enterprise service bus translates between the external and internal context representation (see section 2.5.2). The validation of context information happens in two stages: (1) the service bus has to transform the external context representation into instances of the context types. (2) Rules in the rule engine can further validate the data. Dealing with ambiguity and data of different quality has not been implemented in the prototype and will be addressed in the future.

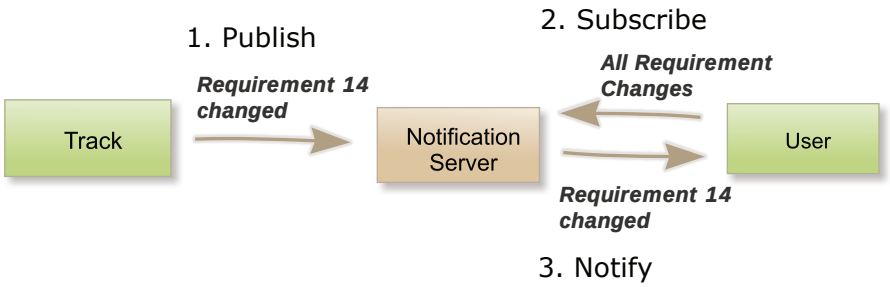

Figure 2.7: Publish-Subscribe Notification System

2.3 Context-aware Notification Systems

The goal of notification systems is to provide users with valuable information in an efficient and effective manner without interrupting the user's primary task [57].

Unnecessary interruptions occur frequently [56], consume a lot of time [80], and degrade performances [79]; thus, they have to be minimized. In NOTICON we tried to minimize the unnecessary interruption by (1) displaying the notification in the tools the user is currently using, (2) allowing the definition of complex rules to reduce the amount of notifications, but increase their importance, and (3) providing a flexible way to specify the content of notifications.

In GSD projects notification systems can be used to:

- Raise the awareness of the availability of other team members.
- Inform about potential conflicts before they become problems.
- Reduce the amount of notifications that have to be sent manually.
- Individually and timely provide users with information they are currently interested in.

The common communication pattern in a notification system is Publish-Subscribe as illustrated in figure 2.7. The publisher publishes events to a notification server regardless of who is interested in the information. Components express their interest in particular events via subscriptions, and the notification server forwards all events

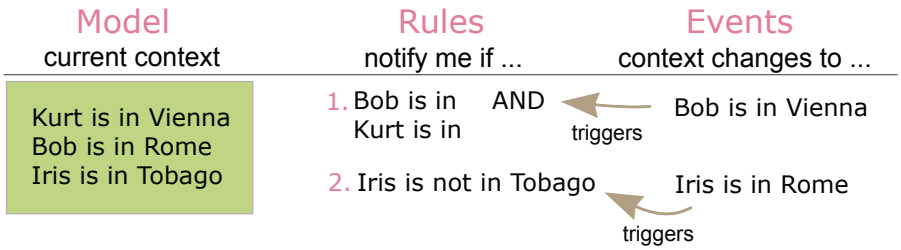

Figure 2.8: Context-aware Notification System

that match the subscription pattern.

In a context-aware notification system, the notification server maintains a context model. Rules define actions that should be executed when the model reaches a specific state; eg. if Bob and Kurt are at the same location, notify Kurt about this fact. Events that are published to the notification server are not forwarded to subscribers, but change the context model - which in turn can trigger rules that create notifications. Subscribes do not subscribe to patterns of events, but define rules when they want to get notified and what the notification content should be.

Figure 2.8 illustrates a context-aware notification system. The current context contains three facts about the current location of the users Bob, Kurt and Iris. Kurt specified two situations in which he wants to get a notification: 1) if he and Bob are at the same place, and 2) if Iris is not in Tobago. A notification is sent, whenever the context changes to a state that matches Kurt's rules.

Section 2.3.1 describes a framework on how notification systems can be described and applies this framework to NOTICON. In section 2.3.2 we shortly introduce existing solutions that influenced our work.

2.3.1 Framework for Notification Systems

Rosenblum and Wolf defined that a notification system "would have the ability to observe the occurrence of events in components, to recognize patterns among such events, and to notify other, interested components about the (patterns of) event occurrence." [70]. They propose a framework that describes notification systems in

seven models. It is useful to understand how NOTICON works:

1. The *object model* characterizes the components that generate events and the components that receive notifications about events. Eg. Eclipse could be both an event generator and a notification receiver.

2. The *event model* describes the events that are sent by the event producers. NOTICON does not enforce a specific format but can integrate any format that a tool may use. Eg. if Rational ClearQuest is just able to send Emails in case of events, we can transform the content of the email into our internal representation at the server side.

3. The *naming model* defines how components refer to other components to express their interest in events. In NOTICON we completely decoupled components from each other, so no direct references between them are necessary. The notification server connects to various event producers simultaneously in different ways (eg. by using message queues, regularly polling a mailbox, etc.)

4. The *observation model* defines how event occurrences are observed and related. The observation could be either system- or user-triggered [78]. In a user-triggered observation, the users is responsible for sending the event (eg. by writing a message in a text box and clicking the "send" button). In a system-triggered observation, the system is responsible for sending the event as soon as it occurs or within an interval (eg. every hour). NOTICON does not differentiate whether an event was system- or user-triggered and treats them uniformly.

5. The *time model* defines the temporal and causal relationships between events and notifications. In NOTICON we support three modes: 1) "immediate" delivers the notifications as soon all conditions of the notification rules are met. 2) "delayed" sends the notification until other conditions in the context are met (eg. a person logs in). 3) "batch" delivers notifications in a bundle according to a predefined schedule.

6. The *notification model* defines the mechanisms that components use to express their interest in receiving notifications. In NOTICON rules observe the context

model and trigger notifications when they match. The notification contains information about the intended receiver (eg. Dominik) and the channel that should be used for delivery (eg. Email).

7. The *resource model* defines where the observation components are located. In the current implementation NOTICON requires one central node that can be reached by the event producers and notification consumers. The message based architecture allows any number of intermediates that could eg. route events between multiple notification servers; a distribution of the context model is not supported yet.

2.3.2 Overview on notification solutions

A wide range of notification systems exist, ranging from domain independent systems to application specific solutions. In this section we will describe the notification systems where we got ideas from or that we utilized or plan to utilize in the future.

Systems like *JMS* [35], *READY* [34], CORBA-NS [2], and *Hermes* [66] are general purpose event notification systems that provide an advanced and comprehensive set of features. Their focus lies on an efficient distribution of events and subscriptions between event consumers, producers, and servers. They try to minimize network traffic by using sophisticated protocols and routing algorithms. On contrary they are often bound to technologies (eg. Java in case of JMS) or complex programming models (eg. CORBA in case of CORBA-NS). ActiveMQ [6] is a JMS compatible open source message broker that supports topic based subscriptions and cross-language clients and protocols. We used it in the prototype to connect with the Eclipse plug-in.

Other notifications services like *Siena* [13] and *Elvin* [75] provide a smaller set of features, but put great emphasizes on fast and scalable routing of events. A simple, but expressive filtering languages is used by the receivers to specify their needs in certain events or sequences of events. YANCEES [23] is a highly extensible event

[6] http://activemq.apache.org

service that builds on existing publish-subscribe infrastructure, but flexibly extends it with plug-ins. It can also integrate different event systems like Siena and provides clients with a uniform subscription language.

CASSIUS [43] is specialized in supporting awareness applications. It can collect awareness information from diverse sources and route it into various awareness tools. *FeedMe* [76] is a collaborative alert filtering system that is based on XML feed protocols such as RSS and ATOM. Users can rate the alerts they receive and by applying machine-learning techniques, FeedMe can infer preferences for future notifications. *Palantir* [72] is built on top of Siena and plugs into configuration management systems like CVS to raise the awareness of workspace changes among developers. The tool informs developers about which other developers change which other artifacts. It integrates very nicely into Eclipse [68] and we plan to provide a similar representation in future versions. *World View* [71] provides a view of the teams and their interdependencies in GSD projects and intends to help developers to identify global and local team members.

To give an overview on how NOTICON differentiates itself from and relates to the systems described above, we compared them along six attributes as shown in table 2.1. Each attribute was rated with a value from 1 to 5 where 5 is generally preferable for a notification solution. The values were estimated based on the articles (see above) that describe a particular solution and differentiate it from other solutions in their related work sections. The six attributes are as follows:

- *Extensibility:* Defines if and how flexible the system can be extend with new functionality. Higher extensibility is good if complex scenarios should be realized. (1 . . . not extensible, 5 . . . very extensible)
- *Portability:* Refers to the ability to interact with the system from various platforms and integrate it into different applications. This is important if it is not known in advance, in which environment the system will be embedded and with which systems it has to interact. (1 . . . not portable, 5 . . . very portable)
- *Performance:* Estimates performance characteristics of the system. This is important to estimate how usable the system may be in large, frequently changing

environments. (1 . . . slow, 5 . . . fast)

- *Learnability:* Estimates how easy the system can be learned and used by developers or users. This is important for a quick introduction into a project. (1 . . . difficult to learn, 5 . . . easy to learn)

- *Focus:* Whether the system is very much focused on a specific task or domain. Systems that are very focused may provide very good usability in their domain, but may not cover less requirements. (1 . . . very much focused on a domain, 5 . . . usable in many domains)

- *Functionality:* How much functionality the system provides out of the box, eg. how complex rules can be defined. (1 . . . limited functionality, 2 . . . very rich functionality)

Table 2.1 shows that NOTICON has the worse rating of the performance attribute. This is because all other solutions can distribute events immediately as they arise and do not have to maintain much state. In NOTICON a context model is maintained on the server that can be very large in big project; thus, we expect the performance to be lower. We did not evaluate yet whether performance issues may become a problem in large projects, but will investigate this issue in future work.

2.4 Event Stream Processing and Rule Engines

Implementing notification system can either be done on top of an event stream processing engine or a rule engine. In NOTICON we decided to use a rule engine and this section will give some background information about stream and rule engines and why we decided for the latter to implement NOTICON.

system	extensibility	portability	performance	learnability	focus	functionality
NOTICON	5	5	1	3	3	5
JMS	1	4	4	4	5	2
CORBA-NS	1	3	5	1	4	3
Siena	3	4	4	4	4	3
Elvin	3	3	4	5	5	2
Yancees	5	4	2	3	4	4
Cassius	4	2	3	2	2	3
Palantir	2	1	3	5	1	1

Table 2.1: Comparison NOTICON to related systems.

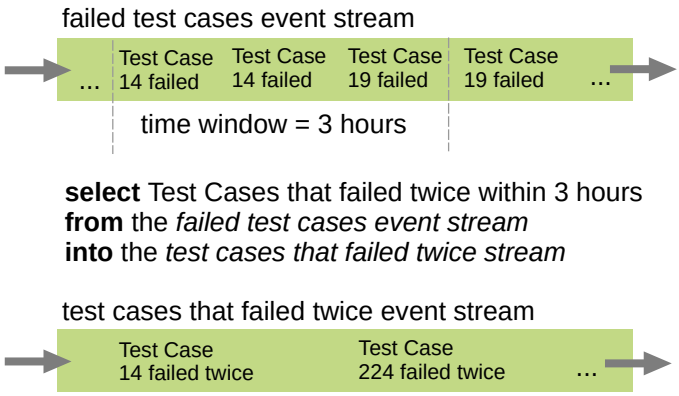

Figure 2.9: Simple event stream processing.

2.4.1 Event Stream Processing

The concept of *Complex Event Processing* (CEP) was coined by David Luckham and refers to the ability to discover complex patterns among multiple streams of events and aggregate them into higher level events that can be consumed by applications [53]. The job of an Event Stream Processing (ESP) engine is "to consume multiple streams of event-oriented data, analyze those events to discover patterns and act on the inferred events it finds - in milliseconds" [62].

Figure 2.9 illustrates on a simple example how event stream processing works: In the "failed test cases event stream" applications publish events about failed test cases as soon as they occur. Another application might only be interested in events that failed twice within a specific period of time, thus the application developer formulates an according stream query that gets interpreted by a stream engine (eg. Esper [7]). A *window* restricts the event stream by certain criteria; eg. to contain only events that occur within three hours. The *pattern* defines sequences of events within the window that should be discovered (eg. two Test Case events with the same ID). When a pattern matches, consequences can be executed (eg. a message is sent to a user). Alternatively the events that matched the pattern can be aggregated into a higher level event (a complex event [53]) and inserted into other event streams as shown in the figure.

[7] http://esper.codehaus.org

Our first architectural approach for the prototype was based on event stream processing because it seemed rather obvious: A variety of systems publish events independently from each other. Users subscribe to event patterns and and receive a notification when the pattern matches. This is the approach that most of the other systems we evaluated use.

When we tried to bring context awareness into play, we quickly discovered the limitations of this approach: Context information (eg. the file a user is currently working on) are not events, but describe situations that are normally valid for a period of time. Thus, formulating patterns on event streams that include context information, turned out to be very complicated for developers and impossible for business users.

As a consequence we dropped our initial design and switched to an approach that builds on a rule engine as described in the next section.

2.4.2 Rule Engine

In NOTICON we use the Java based rule engine Drools [8] which is more specifically defined as a forward-chaining production rule system.

The basic principle of a production rule system is relatively simple: a production (or rule) consists of a condition and an action. The condition contains a set of propositions that can be either true or false. If all propositions in a rule are true, the rule is said to be activated and the action can get executed.

```
1 IF
2   1 == 1
3 THEN
4   print "Hello World"
```

Listing 2.1: Simple production rule

Listing 2.1 shows a very simple production rule: if 1 equals 1, the "Hello World"

[8] http://labs.jboss.com/drools/

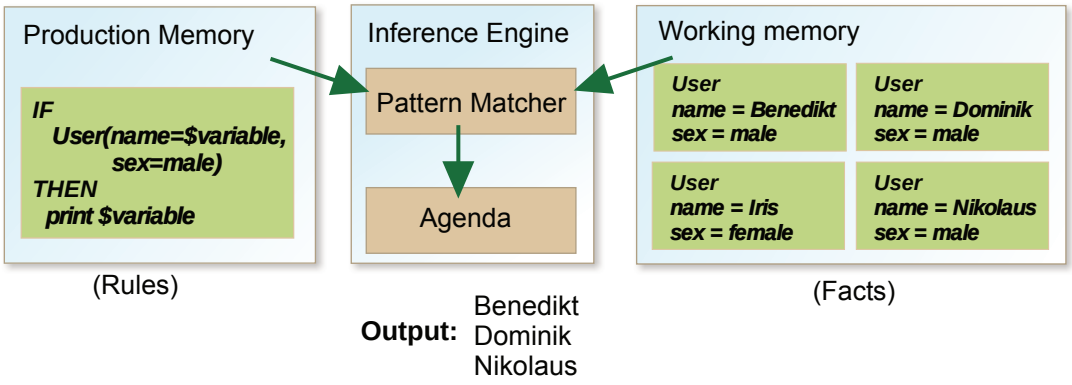

Figure 2.10: Components of the Drools Rule Engine.

should be printed. Because all propositions are true, the rule is activated and the action can be executed.

Propositions are not constant statements in most cases, but rather use facts that are maintained in the *working memory* of the rule engine as shown in figure 2.10. A *fact* is a statement about something that is true (eg. Benedikt is male). They are represented as Java object in Drools. In the example shown in the figure, four facts, represented as objects of type "User", were inserted into the working memory. A rule that matches all male users was inserted into the *production memory*. The name property of all male users is bound to the `$variable` variable. If there would be a second proposition in the rule that also refers to the variable, its value would not be overwritten, but treated like a constant. In our example, three objects match the proposition; thus, the rule would get activated three times - each time with a different value bound to the `$variable` variable.

The *pattern matcher* is responsible for matching the rules from the production memory to the facts from the working memory. Drools uses the RETE algorithm [26] to optimize the performance of the pattern matcher. An *agenda* controls the order in which the rules are matched and the activated rules are fired. This is important in cases where rules do not just print statements like in our example, but modify, retract, or add new objects to the working memory. These modifications can lead to the activation of new rules or to a deactivation of already activated ones. The agenda can either be controlled automatically by the rule engine or by the developer.

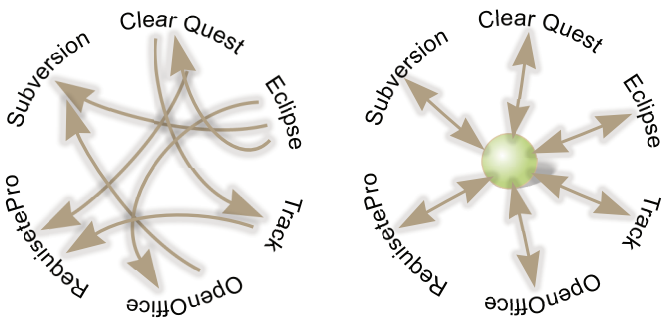

Figure 2.11: Point to Point Integration vs. Integration Bus

In NOTICON , the project context is maintained in the working memory to be able to define notifications that are sent depending on the context. Events that are sent by external systems lead to an insertion, deletion, or modification of objects in the working memory, which in turn can trigger the activation of rules. We defined a domain specific language that allows users define notification rules more easily. Under the hood, notification specified in this language are transformed into Drools rules and added to the production memory.

The next section will describe how NOTICON interacts with other systems.

2.5 Enterprise Application Integration

Enterprise Application Integration (EAI) "has to deal with multiple applications running on multiple platforms in different locations, making the term simple integration pretty much an oxymoron." [41]. NOTICON has to integrate data and functionality of various enterprise applications to (1) gather context information and to (2) display notifications to project members. This section elaborates on the patterns and technologies that are used for EAI and that NOTICON is built upon (see chapter 4).

The goal of EAI is to connect heterogeneous systems to realize functionality that cannot be provided by one application alone. Most business processes span several applications and very often users have to maintain the same data in several sys-

tems. EAI allows both data and logic integration. With EAI organizations can gain competitive advantage by automating business processes and reducing errors that happen with manual integration.

The left side of the figure 2.11 illustrates *point-to-point integration* where direct communication is established between two applications. The leads to very tight coupling between the applications, and the number of connections that have to be created is high. If n applications needed to be integrated with each other, $\frac{n(n-1)}{2}$ connections would be required. For the six applications shown in figure 2.11 fifteen connections were needed. The tight coupling between the application results in a hardly maintainable system and replacing applications becomes nearly impossible. Also the reliability of the whole system decreases as applications depend on each others functionality.

The right side of figure 2.11 shows the integration scenario via a message based integration bus. Applications do not communicate directly with each other, but only exchange messages with the message bus, which is responsible for mediating the information flow between the connected systems.

Service Oriented Architectures (SOA) based on *web services* are the foundation for successful integration projects. Components describe the functions they can perform in the technological independent WSDL format, and data is transferred according to the SOAP protocol encoded as XML. Due to the separation of service description and service implementation, a service consumer and service provider are only loosely coupled, and both of them could be replaced without requiring changes of the other.

An Enterprise Service Bus (ESB) further decouples service consumers from service providers. It mediates the message flow between the services and provides a reliable asynchronous connection between them. An ESB can also transform messages and queue them in case the receiver is not available.

ESBs are the base for systems based on an Event Driven Architecture like NOTICON. In such an architecture there are no methods that get invoked on services and that return immediate results, but instead components publish events to an "event cloud" and subscribe to events in the cloud that they want to react upon.

Section 2.5.1 will elaborate more on SOAs and briefly describe how web services work. Section 2.5.2 will describe the functions of an ESB and the message patterns that it is built upon. In section we will elaborate on the characteristics of an EDA.

2.5.1 Service Oriented Architecture

Service Oriented Architectures gained a lot of attention within the business and IT community. The Oasis Group defines a service oriented architecture as "a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains." [55]. Service providers typically offer whole business processes like "process incoming order".

SOAs have the following characteristics [14]:

- All functions in a SOA (business, or technical) are defined as services.
- All services are independent, and callers do not have to know how a service performs its function.
- The location of the component that provides the service is transparent to the caller.

The key to provide these characteristics is the separation of the service interface from the service implementation. The interface defines the required parameters and the nature of the result in a technological independent way.

Web Services are currently the most used technology to realize a SOA as they provide maximum interoperability.[64]

Web Services

SOAs based on Web Services are commonly used for EAI. The protocol for exchanging data is SOAP. It "provides the definition of the XML-based information which can be used for exchanging structured and typed information between peers in a

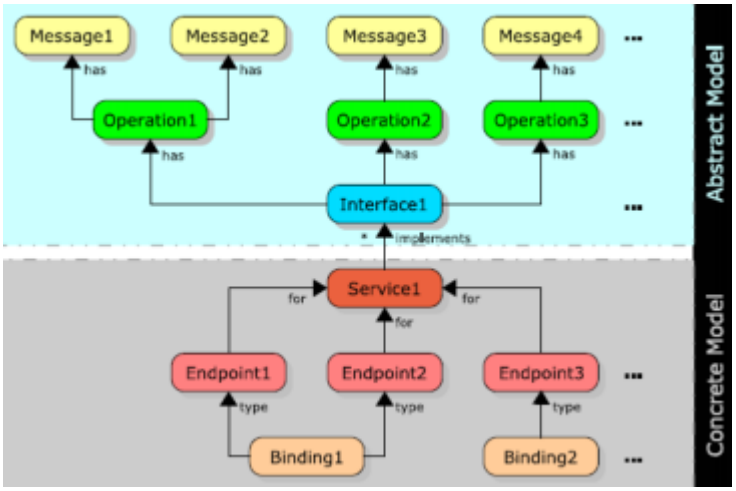

Figure 2.12: Structure of a WSDL Service Description [82]

decentralized, distributed environment." [82] A SOAP message (SOAP envelope) consists of a *Header* and a *Body* part.

- The *Header* contains metadata, like the operation that should be called on a service, how the payload has been encrypted, and where the response should be sent.
- The *Body* contains the actual data that should be transferred, encoded in XML.

The functionality and application provides (the service) is described in the technological neutral XML based WSDL format. In WSDL2 a service is described in two fundamental stages [15] as illustrated in figure 2.12:

- At the *abstract level* the messages that the Web service sends and receives are described in a wire independent format (eg. XML Schema [22] or OWL [58]). Message exchange patterns describe the message flow (eg. "In-only" or "Request-Response"), and operations associate messages with message exchange patterns. Interfaces logically group operations.
- A the *concrete level*, bindings specify the wire format and transport protocol for transmitting messages of an interfaces, eg. SOAP over HTTP. An endpoint

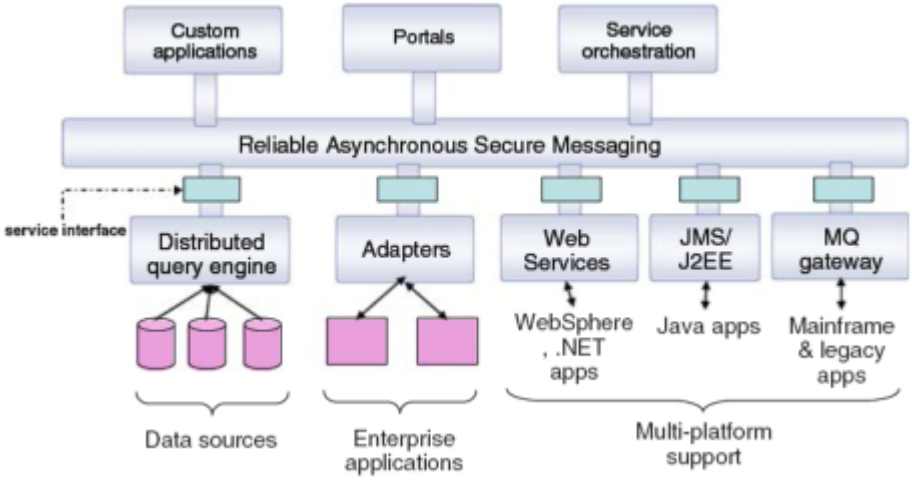

Figure 2.13: Enterprise Service Bus [64]

associates bindings with network addresses, and services group endpoints that implement a common interface.

2.5.2 Enterprise Service Bus

A service oriented architecture based on Web Services does not necessarily lead to the desired loosely coupled message based integration (see figure 2.11). This can be archived with an ESB which acts as a Mediator [29] between services (see figure 2.13). An ESB is "an open, standards-based message bus designed to enable the implementation, deployment, and management of SOA-based solutions with a focus on assembling, deploying, and managing distributed SOA" [64]. It provides the backbone of a SAO and is responsible for orchestrating the message flow between services.

Figure 2.13 shows that an ESB can connect systems regardless of their technology or the transport protocol that is used to exchange messages. The only requirement of a connected system is that it provides a service interface. The ESB takes care of the routing and transformation of the messages between the services. The JSR-208 specifies how an Enterprise Service Bus on the Java platform should look like. For NOTICON we decided to use the JSR-208 compatible open source message bus

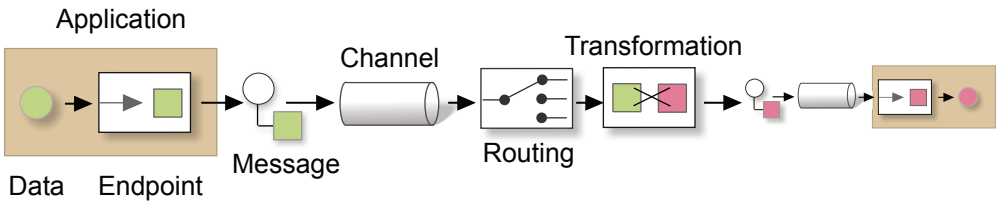

Figure 2.14: Message based integration

"Mule" [9].

An ESB encourages highly asynchronous communication patterns between services. Components should not call service methods and block till the answer arrives because the component that implements the service might not be reachable. Instead a service operation should be specified that gets called when the results are available. The service bus takes care that the messages are routed to the correct services and can delay delivery if the service is not working.

Figure 2.14 illustrate flow of data in a message based ESB [41]:

1. Most application do not have the built-in capability to use messaging systems like JMS or ActiveMQ. Thus they interact with an *Endpoint* that knows how it can invoke functions on the application and how to exchange messages with the service bus.

2. The endpoint wraps the data that the application wants to send in a *message* and optionally adds additional metadata to it.

3. Messages are transmitted via *channels* that establish a reliable asynchronous connection between sender and receiver. The sender puts the message into the channel and the messaging system (eg. ActiveMQ or BizTalk) takes care that it can be picked up by the receiver. It is important to note that the message system does not guarantee immediate delivery, but guarantees delivery. This is most often an advantage and makes message based integration very stable even if integrated applications are temporarily not available.

[9] http://mule.codehaus.org

4. The message bus is responsible for delivering the message to the receiver. Based on a pipe and filter architecture, the message is routed through various components that can process the message. *Routers* are components that route the message based on criteria like the payload to other components or the final receiver.

5. *Transformers* are responsible to transform the message payload to other formats. This is required if the integrated applications have not agreed on a common data format.

2.5.3 Event Driven Architecture

Service Oriented Architectures based on an Enterprise Service Bus are good for Enterprise Application Integration, but often resemble a call-stack driven architecture that exhibits the following characteristics [40]:

- *On thing happens at a time:* There is normally a single line of execution where one thing happens after the other. Concurrent execution of tasks is possible, but has to be treated carefully.

- *The order of execution is known:* Because one method calls another, the caller has a good idea of what should happen next. With BPEL [42] services can be combined regardless of their implementation, and the call-stack is described declaratively in XML. Nevertheless describes BPEL a process and a such implies a sequential execution.

- *It is known who can provide a needed function:* The caller has to know that there is another component that can provide the function it wants to call. With Web Services and an Enterprise Service Bus, the caller does not need to care about the concrete component but still needs to know the service interface.

For a notification system, this architecture would require all connected systems to be aware of each other and invoke each others operations appropriately. In case of a new notification, the ESB would have to lookup the service endpoints of the applications

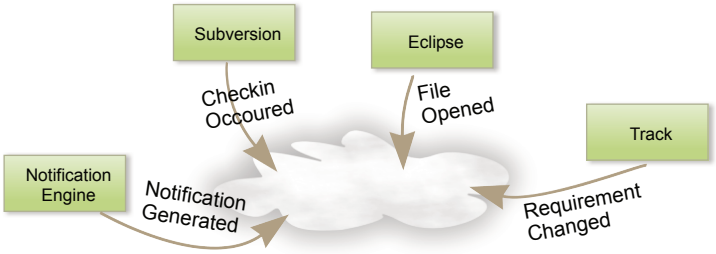

Figure 2.15: Event Cloud

that should receive the notification and call methods like `showNotification()`. Fairly complex coordination logic would be required to realize this process.

In an Event Driven Architecture, communication between components is not by established by calling methods, but by exchanging events. An event "signifies, or is a record of, an activity that has happened" [53]. Luckham [53] coined the term "Event Cloud" to explain the principles of an Event Driven Architecture (see figure 2.15. Applications send events into the Event Cloud and register for events they are interested in. Eg. Subversion sends a "Commit Occurred" into the Event Cloud as soon as a commit occurs. Another component (eg. Eclipse) could react on this event, do some processing and publish the results again as events.

Integrating applications with an EDA is a fundamental shift of responsibilities. "It allows components to be decoupled to the extent that the 'caller' is no longer aware of what function is executed next nor which component is executing it." [40].

An Enterprise Service Bus can be used as an Event Cloud. Services would just register the events that they provide and want to consume with the service bus, which is responsible for managing the subscriptions and routing the events between the components.

The architecture of NOTICON is an EDA because we wanted it to be highly extensible and provide a platform where additional services (eg. event monitoring, build servers, etc.) could be plugged in easily.

3 Research Issues

In section 3.1 we describe our research questions; how we answered them is described in section 3.2. To gather the requirements for a solution we performed a detailed stakeholder analysis which is described in section 3.3. Section 3.4 elaborates on the requirements that we deduced from the stakeholder analysis. In section 3.5 we describe the notification discovery and description framework and process that can be used by project managers to discover communication risks and analyze how they can be mitigated with a notification solution.

3.1 Research Questions

Our research questions were:

- *What are the key requirements of a notification system in the context of GSD?* We performed a stakeholder analysis to discover and describe the most important requirements of a notification solution in global software development projects. To our knowledge there is no publication of requirements for a notification system based on a stakeholder analysis, and existing systems are more driven by technical challenges or specific use cases. Some important requirements may have been missed, but are critical for a successful introduction of a notification solution in a large scale project.

- *How can the notification needs in a GSD project be discovered and described?* Arguing that notifications are required, is not enough: A method has to be provided to discover the concrete notification needs and formally describe them.

To our knowledge there is no applicable process that guides project manager through introducing a notification system. We propose a process that has been specifically designed to be applicable in a constrained environment (low budget, few time) and that guides from a general communication risk analysis to a formal definition of the notifications.

- *What architecture fulfills the requirements for a notification solution in the context of GSD?* Existing solutions as described in section 2.3.2 are either designed for very specific use cases (eg. make users aware of each other in Eclipse when they are working on the same source file) or provide very generic functionality that is not readily applicable to notification scenarios in GSD projects. An extensible platform with rich functionality but well aligned to the GSD domain is missing. We propose the prototype NOTICON that provides advanced notification capabilities and that can be easily tailored to the specific needs and infrastructures of a GSD project.

- *How can we validate our process and the usefulness of* NOTICON*?* In cooperation with Siemens PSE we did an empirical evaluation of both the process and the prototype. Based on a simple but realistic project setting we show how we applied the process and configured the prototype for specific notification scenarios.

3.2 Research Design

The first step was to get an overview on GSD projects and their characteristics with a detailed literature research. Then a stakeholder analysis was performed to gather the requirements for a notification solution. Based on these requirements, existing solutions were evaluated. After that, the prototype NOTICON was developed. The notification process and the setting for the case study were developed next and refined with experts from Siemens PSE. The last step was to test the prototype and the process on a small feasibility study.

Domain of inquiry	Goal stakeholder for suprasystem	Means stakeholder for Suprasystem
	Goal stakeholder for "System under Consideration" (SuC)	Means stakeholder for SuC

Figure 3.1: Generic stakeholder classification layout

3.3 Stakeholder Analysis

Complex IT systems have a wide variety of stakeholders with different and sometimes contradicting interests. "A stakeholder is a person or organization who influences a system's requirements or who is impacted by that system" [31].

In section 3.3.1 we describe the stakeholder classification framework by Preiss and Wegmann [67] that we used to discover the most important stakeholders. In table 3.1 we describe the major value of NOTICON for each stakeholder, their attitudes toward the system, and their major interests. A detailed description will be given in section 3.3.2.

3.3.1 Stakeholder Classification Framework

The stakeholder classification framework by Preiss and Wegmann [67] is centered around the concept of systems. The *System under Consideration* is the system that is going to be built and that gets embedded into an existing system - the *Suprasystem*. Both the goals of a system and the means to archive these goals must be analyzed. The domain is partitioned into the *systems development* domain and the *systems operation* domain. The system development domain include all development activities (plan, design, implementation, etc.). The system operation domain is about the runtime of the system in a real environment as well as the management of change.

Figure 3.1 illustrates the general layout of the stakeholder classification framework.

Stakeholder	Major Value	Attitudes	Major Interests
OS Community	new commercially usable open source project; shows usefulness of open source in business contexts;	positive but no involvement at the beginning	stability of the project; clean implementation to get ideas for other projects; many organizations using it, to raise the awareness of open source software for businesses in general
Tool Providers	leverage existing infrastructure;	positive if the integration cost are low and the company does not have a proprietary notification solution in its portfolio;	simple API; easy integration; no replacement of own tools and applications; no performance impact on the companies products
NOTICON Developers	commercial models; challenge; reputation	positive	easy to extend; easy to understand; well known in the OS community; clean code;
System Administrators	can be used for monitoring systems	more likely to be negative because of extra work	not much additional work; upgrades of other systems must not cause problems; no negative performance impact; no security issues;
Executives	higher revenues	positive if $revenue > effort$	low introduction risk; fast return on investment; quick introduction;
Project Managers	higher productivity; less errors because of communication mistakes; reduced risk;	very positive because of the expected productivity gain	powerful notification rule capabilities; good usability; metrics and reports; quick system introduction;
Developers	reduced integration effort; better general information and awareness;	positive if no extra effort for them	non intrusive notification; personalization; usability; privacy concerns

Table 3.1: Stakeholder summary

Domain of inquiry	Name of the system	Goal Stakeholder	Means stakeholder
Development	Suprasystem: Open Source Community	Community Members; Tool Provider;	OS Community members;
	SuC: Development project	other OS project members that built upon the core compnents; Hosting Provider (eg. Sourceforge)	Students; Developers of other OS projects; Scientific Community; Developers;
Operation	Suprasystem: Company running the notification system	shareholders; customers; software and hardware suppliers;	Management; Employees; IT Department
	SuC: The notification system	project management; developers of the GSD project; quality assurance; testers; information gatekeepers; software suppliers;	developers of the GSD project; system administrators; helpdesk; tool vendors

Figure 3.2: Stakeholder Classification

The goal stakeholder for the suprasystem are interested in the perceived behavior of the suprasystem only. The mean stakeholder for the suprasystem are interested how the suprasystem archives its behavior by looking a the structure and the interconnection of the inner systems. The goal stakeholders for the System under Consideration (SuC) are interested in the perceived behavior of the system. The means stakeholder for the SuC are interested in the way the SuC archives its behavior.

3.3.2 Stakeholders

Figure 3.2 shows the stakeholders classified by Preiss and Wegmanns [67] classification framework. The development domain contains all stakeholders that are concerned with open source development in general (the Open Source Community Suprasystem) and all stakeholders that are directly involved in planning and developing NOTICON. In the operation domain the suprasystem is the company running NOTICON and the system under consideration is NOTICON itself.

The following sections describe the most important stakeholders in greater detail.

Open Source Community

Description The OS community includes all people that actively participate in open source projects.

Major Values If NOTICON is successful, it would steady the claim about the usefulness of open source software in general. Also the ideas of NOTICON and NOTICON itself can be leveraged in other open source projects.

Attitudes The attitudes toward the project are very positive although no involvement is expected unless the application is successfully deployed in one or more companies.

Involvement In the future the OS community should take over the project, maintain, and extend it. Adapters for legacy systems are expected to be developed by the OS community.

Major Interests An important interest of the OS community is the stability of the project. Projects that pop up and disappear after a month harm the reputation of OS software as a reliable and trustworthy investment. The community is also interested in well organized source code, a simple API, and good development documentation so that interested persons can easily join the project and get started quickly.

Success Criteria (1) The project is actively maintained for longer than the average number of open source projects. (2) It does not take longer than a few hours to understand how to system works and be able to write extensions.

Tool Providers

Description Tool providers are software producers like IBM or Mercury which sell software that is likely to be integrated into NOTICON.

Major Values Integrating with NOTICON makes the tool more valuable for companies that use NOTICON. If a product does not provide notification mechanisms,

it can leverage an existing infrastructure.

Attitudes The attitudes are positive as long as the integration efforts are not too high and the value of the own product increases. They might be negative if the tool provider already has another notification solution embedded in its products and does not want to integrate with other applications.

Involvement As long as NOTICON is not used by several big companies, it is very unlikely that tool providers will be actively involved in the development of NOTICON; though, it is expected that they provide APIs and documentations for their products, which can be used by the OS community for integration.

Major Interests The system must not interfere with the activity of their software as this may harm the tools providers reputation. In case a tool provider wants to integrate with NOTICON, its API must be as simple as possible and integration must not require much effort.

Success Criteria An integration is successful if the functionality of the integrated application is not touched at all, and the performance is not decreased so much that it can be noticed by users.

NOTICON Developers

Description The developers that are responsible for creating and maintaining the systems core and the legacy system adapters.

Major Values Developers may benefit from NOTICON by adopting a business model and earn money with it. Participating in an open source project also increases a developers reputation, and the challenging tasks improve individual skills like programming, teamwork and communication.

Attitudes The developers attitudes are very positive toward the project.

Involvement Their involvement is necessarily a lot.

Major Interests The project should be known in the community - otherwise the

reputation benefits would not arise. It has be structured in a way that allows many persons working on it at the same time. Developers might vote for a restrictive license because they might not want other companies making money with it.

Success Criteria The project should be well known in the OS community. This criteria can be measured by the number of awards the project receives (eg. Source Forge's "Project of the Month").

System Administrators

Description System administrators subsume people in the IT departments which maintain the IT landscape of the company.

Major Values They can benefit from NOTICON if they use it for their own processes (eg. system monitoring).

Attitudes The attitudes of the system administrators toward NOTICON are expected to be rather negative. It is likely that the installation will require modification on many machines and programs. Also a completely new system has to be maintained and supported which increases the workload of the department. If the administrators can efficiently use NOTICON for system monitoring, the attitudes are expected to be positive.

Involvement Although the attitudes are likely to be negative, the system administrators are key stakeholders. They have to roll out the system to the machines, provide the necessary hardware, maintain and support the system, and react in case of problems.

Major Interests The system will increase the workload for system administrators, thus the additional work has to be kept to a minimum. The installation must be as easy as possible (and probably supported by extern consultants) and upgrades of other systems must not cause problems for NOTICON. Maintenance should be barely required and runtime configuration can be done by the users. System administrators demand that the notification system does not have a

negative performance impact on other systems and that problems in NOTICON do not cascade to other systems. It also must not weaken the existing security infrastructure, and it must not require additional investments in it. The system administrators also require a thorough documentation.

Success Criteria (1) The installation can be done within a day. (2) Updates do not take longer than the average of the other systems. (3) The system does not need any attention, as long as there are no fundamental changes in systems that NOTICON integrates.

Executives

Description Executives manage and control the enterprise and are responsible for the financial success.

Major Values Executives expect higher revenues because of reduced communication costs and reduced costs caused by rework and errors.

Attitudes The attitudes are positive as long as the expected revenues justify the additional efforts.

Involvement The support of the executives is very important for the success of NOTICON. Without that, it is unlikely that the other stakeholders (especially the system administrators) will take the extra effort that is required in the introduction phase of NOTICON.

Major Interests The risk for introducing the system in the organization must be low and a high revenue is expected. The return on investment has to be very short (within a year) and the installation costs kept to a minimum. Furthermore, it must be possible to quickly introduce NOTICON in a smaller projects and scale it to larger ones if successful. Support and continuous improvement is also of great interest and management must be able to calculate the costs for adaption.

Success Criteria The return on investment must be less than the project duration.

Project Managers

Description Project managers are responsible for the successful realization of a project within a company. They normally have a limited budged, but decide more or less freely on its use.

Major Values Project managers expect a higher productivity (eg. measured in lines of code or function points), less communication defects, and a communication risk reduction.

Attitudes Their attitudes are very positive toward NOTICON .

Involvement The involvement of project managers needs to be high, because it is in their responsibility to communicate the need for such a system and motivate the other stakeholders to use the system. They know many of the communication needs of the project, thus will define many of the notification rules.

Major Interests Project managers will have very demanding requests towards the functionality of NOTICON . They expect that highly complicated rules can be specified efficiently and the notification specification can be communicated easily. Project managers also expect metrics and reports to argue the need for NOTICON with the executives. Especially in GSD projects, where several project managers are working together, they have a strong interest that the systems can be introduced independently from each other. If another subproject also decides to use NOTICON , the integration has to be seamless.

Success Criteria The number of communication defects must be reduced by the amount that has been set in advance. This can be measured eg. by comparing the average number of hours it takes to implement a change before and after NOTICON is introduced.

Developers

Description Developers develop the GSD systems components. Depending on the organizational structure, they get their tasks assigned by some group leaders, or they can decide the next requirement that they are going to implement themselves.

Major Values Developers expect a reduced integration effort that arises due to communication problems. Integration efforts are tasks like merging source code that has been worked on concurrently, or rolling back to a previous version because a requirement has changed. They also expect better information about what is going on in the project and better awareness about what the other project members are doing.

Attitudes Developers are expected to have positive attitudes toward NOTICON, as long as no extra effort is required by them and privacy concerns are addressed.

Involvement Their involvement is very high because they are the primary receivers of the notifications. It is also expected that developers create notification rules themself or adapt them to meet their requirements.

Major Interests It is important that no extra work is required for the developers to use the system. Furthermore, they have a strong interest that the notifications do not interrupt their current activities, and that they receive less notification emails from existing systems. Developers want to personalize the message content and be able to specify notification rules on their own. Privacy issues are also very important for them: the system must not allow managers to control what they are currently doing (eg. by defining a notification that just shows the current context of a user).

Success Criteria The number notification messages sent by email decreased significantly. The time that is spent fixing code because of communication errors is reduced. Concrete measurements should be defined before introducing NOTICON and evaluated regularity.

3.4 Requirements for the prototype

This section presents the most important requirements for a notification system based on the stakeholder analysis described before. In particular the socio-technical requirements are listed because they are easily overlooked by technicians.

- **The system must be built upon standard technology.** Administrators can maintain a system much easier if they are already familiar with the underlaying technologies. As an additional advantage a wide variety of support infrastructure (documentation, forums, chats) exists that can be accessed by administrators in case they want to extend the system or have an infrastructure problem. By using standard technologies, the security requirements of the organization can be met easier because the used technologies are verified and updated continuously by the research community. It also lowers the entrance barriers for new developers which is important for a vital open source project. Furthermore, the open source community profits because OS technologies get introduced in enterprises that might leverage them for other projects as well.

- **The system must support the users work without interfering with their primary tasks.** This requirement is very important for the acceptance of the system. If developers have to have another tool running on their workstation that consumes memory and slows down their work by steadily interruptions, the costs exceeds the benefits. The notification tool must silently integrate into the already existing tool-set and display its messages in an unobtrusive way. For example if a developer is currently working on a piece of code in Eclipse, notification messages should be displayed in this tool and not delivered by email.

- **False positives and false negatives must be minimized.** False positives are information that is delivered to a user, but that is irrelevant for him or her at the moment. Sensing and interpreting the users current context is therefore required. Eg. a information concerning a piece of code X should be displayed only when the user cares about it (eg. by editing X in Eclipse). False negatives are notifications that should be delivered to a user but are not.

Eg. an information that two developers A and B are changing the same piece of code at the same time is delivered to A but not to B. The impact of false positives is often less than that of false negatives (eg. recognizing and deleting an spam email message only takes a few seconds), but the right balance has to be found. The rules controlling the system must be flexible enough to allow different configurations based on the scenario requirements.

- **The definition of complex notification rules must be supported.** The rule engine must support the definition of rules for complex notification scenarios that are derived from formalized key communications of collaboration processes [85]. It must be possible to reason over data from heterogeneous sources and allow to define to *whom*, *how*, *when*, in which *context*, due to which *event*, with which *content* a notification should be sent and what to do in case of an *error* [84].

  - *who*: The receiver of a notification can be a single person or a group. It must also be possible to specify the receiver dynamically (eg. "all people currently working on requirement X", "all project managers that are not on holiday").

  - *how*: There are different possibilities on how the receivers could be notified, eg. "in the tool they are currently using", "by email", or "by sms". We call this information channels. New notification channels should be easily addable by administrators.

  - *when*: Although most notifications are likely to be sent immediately it must be possible to define different delivery times, like at a specific time every hour/day or condition that has to become true (eg. the receiver comes back from holiday).

  - *context*: To minimize wrong positives, the current context of the receivers must be included in the rule as well. This leads to the derived requirement that the current user context gets sensed and refreshed constantly.

  - *event*: The event (or sequence of events) that triggers the notifications

must be specified, eg. requirement X has changed.

- *content*: It must be possible to specify the content for the notification and include data from different sources. Eg. the notification that another developer is currently working on the same source file should include his or her name and phone number.

- *error*: If a rule cannot be delivered to a person within a specific timeframe or because of an error, escalation mechanisms must be configurable [63]. Eg. the system could retry delivering the message five time and forward it to a colleague after that.

- **Users must be able to specify the notification requirements during runtime.** As discovered in the stakeholder analysis, different people have different notification requirements and want to define and change them themselves. This leads to the technical requirements that notification rules can be changed during runtime. The bigger impact is that this requirement forces the notification system to provide a very usable interface that provides and easy customization.

3.5 Notification Discovery and Description Framework and Process

A notification system deeply integrates into a socio-technical environment and should mitigate communication risks within that environment. To gain the full benefit of a notification solution the project environment be analyzed, and communication risks that can be mitigated with notifications must be discovered and described.

In this section we prose an iterative process shown in figure 3.3 that can be followed by project managers and team leads to discover project communication risks and formalize notifications that mitigate these risks. The process was designed to be easily applicable in practice with tools that are commonly available (eg. an UML

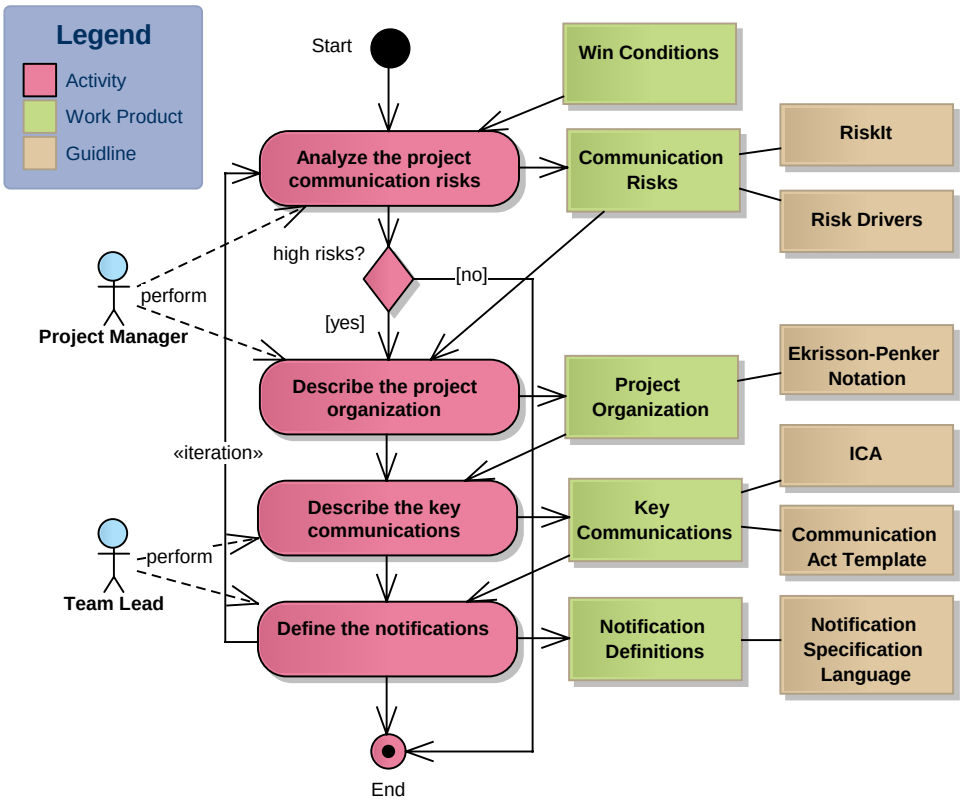

Figure 3.3: Introduction Process

editor). The process should be tailored to the needs of a particular project and not applied blindly [49]; the methods and tools that the project manager is already familiar with (eg. risk analysis processes) should be used wherever possible to lower the entrance hurdles.

3.5.1 Step 1: Analyze the project communication risks

Goal: To find out if there are high communication risks in a particular project that need to be mitigated.

Rational: Mitigating communication risks is not required for all development projects. Eg. if all developers are sitting in the same room and working at the same time, synchronous communication outperforms any technical notification solutions in most cases. Project manager have to judge whether the impact of the communication risks on the win conditions of the stakeholders [11] justifies investments in a notification solution.

Actors: Project managers with assistances from team leads (or their proxies).

Input: The win conditions of the project stakeholders.

Output: A document describing the projects communication risks and the decision to either ignore them or try to mitigate them with tools like NOTICON.

Description: Based on our research of the characteristics of GSD projects (see section 2.1) we discovered a set of drivers for communication risks. For each of those the project manager should (see figure 3.4):

1. Brainstorm on concrete risks that are caused by the risk driver.

2. Rate the relevancy of the driver as A (very relevant), B (relevant), or C (irrelevant). The rating should be based on the impact of the risks on the win conditions of the stakeholders.

3. Perform a more detailed risk analysis if the rating is A or ignore the risk driver

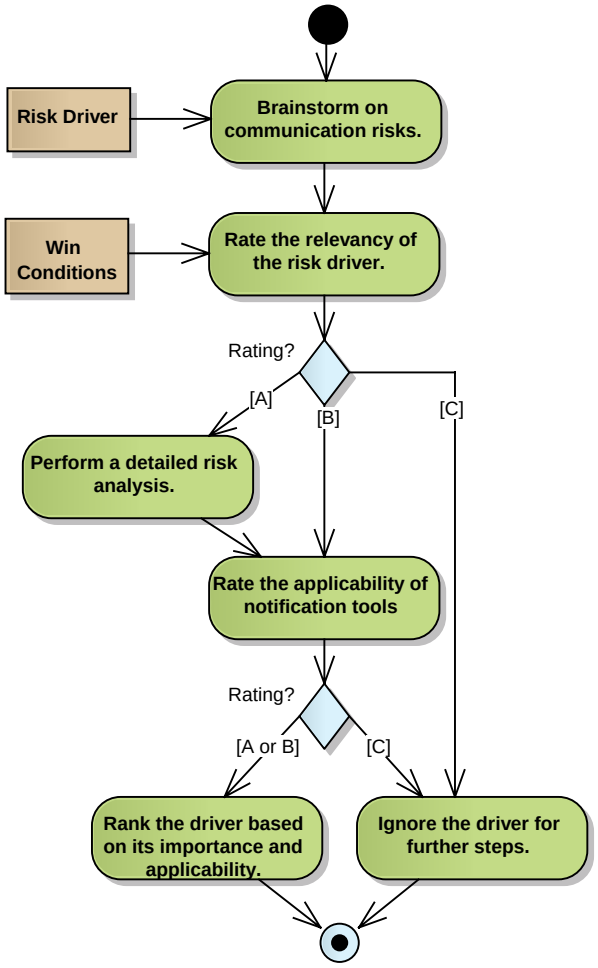

Figure 3.4: Process to rank risk drivers for further process steps.

in further process steps if the rating is C.

4. Rate the driver on how applicable tools (in particular notification tools) are for mitigating the risks with A (very applicable), B (applicable), or C (not applicable).

5. Rank the driver based on its impact and the applicability of tools to mitigate the risks if the rating is A or B, and ignore the driver if the rating is C.

Communication Risk Drivers

The following list will describe each of the risk drivers and give an example to which concrete risks they can lead. We assigned a name to each driver to reference them more easily in subsequent sections, and we roughly ordered them based on the applicability of notification solutions:

- *dependencies: Dependencies among elements that are managed by different teams.* Example: WSDL contracts are maintained by the COBOL team, but are also referenced by components managed by the Java team. There is the risk that changes of the WSDL are not communicated to the Java team and just detected at integration time. The more loose coupled the elements are, the later the changes are often discovered because referenced components can be mocked during development. NOTICON could inform a developer if he or she is working on an element that is used by other teams.

- *changes: Frequent changes of elements that impact several teams at once.* Example: A change in a requirement influences components managed by the COBOL team and components managed by the Java team. If the two teams are not aware of each other, they may change their components independently from each other making them incompatible. NOTICON can raise the awareness of interdependencies between teams.

- *geography: Geographical separation of teams.* Example: The requirements management and the Java development team are located in Vienna, and the COBOL development team is located in Madrid. There is the risk that requirement changes are communicated informally to Java development team (eg. during lunch breaks) and not at all to the COBOL team. This can lead to inefficiencies and frustration within the COBOL team. NOTICON could ensure that changes are communicated in time to both the COBOL and the Java team.

- *formality: Lack of formally defined processes on how changes need to be communicated.* Example: If there is no formally defined process whom to notify in case of a requirement change, there is the risk that the change might not

be communicated at all. Any notification system reduces this risk by forcing a formal definition on which events should trigger which notifications.

- *organizations: Multiple organizations participating in the development.* Example: If more companies are participating in the project, there might not be just one where all requirements are managed. Instead requirements are constantly synchronized between different systems (often manually). There is a high risk that the change of a requirement in one system is not communicated in time to the people managing the requirement in the other system. NOTICON can provide a shared view on all requirements, detect missing synchronizations, and inform interested parties about it.

- *agility: An agile development process.* Example: Agile development processes (like SCRUM or RUP) normally allow frequent changes of components. Regular meetings and continuous integration try to ensure that the changes are communicated to all affected persons in time. For GSD projects the processes have to be adapted and much care has to taken to guarantee efficient communication between teams. Otherwise there is the high risk that changes are communicated within, but not between teams, which lead to all kind of problems. Notification systems can help to apply agile processes in the GSD projects by ensuring the communication of changes between teams.

- *culture: Different cultural background of teams.* Example: People from different cultures can have different communication habits. When communication is necessary across cultures the probability of misunderstandings and offending the other is higher. Emails like "We change requirement X. best regards. Peter" are usual in German speaking countries, but might be perceived as impolite in Japan. Notification system have a limited capability to support communication across cultures, but the text of a notification could be at least altered to match the communication preferences of the receiver. NOTICON could also hook into email programs like MS Outlook and inform the sender of the communication habits of the receiver before the email gets sent.

- *language: Multiple languages spoken within the project.* Example: If Fritz has to inform Mali about a particular situation, but both speak different languages

and only very bad English, misunderstandings are likely to occur. With NOTICON some of those situation can be automated and notifications translated to the mother tongue of the receivers.

If a risk driver seems to be very important for the project, we propose to investigate and document the risks with a more formal method like Riskit. The next section will give a brief overview of this method:

The Riskit Method

Addison and Vallabh show [8] that risk management techniques can reduce the risks in software development projects. But "some of the problems in implementing risk management were the result of difficulties in identifying and quantifying risk since these processes are difficult to translate into reality" [61]. The Riskit method proposed by Kontio [46, 47] is a well known technique and has been validated several times in practice [48, 28]. It is suitable for a coarse grained communication risk analysis which is the goal of this process step.

The most important step in the Riskit method is the creation of analysis graphs that visualize the project risks. They consist of *risk factor, risk event, risk outcome, risk reaction, risk effect*, and *utility loss* elements [46]:

- The *Risk factor* is a fact that influences the probability of risk events. It describes the system environment. Eg. "unstable requirements", or "more than 100 developers".
- A *Risk event* is something negative happening in the project. Its probability is influenced by risk factors, reactions, and other risk events. Eg. "conflicts when merging the source code", "work is done twice".
- The *Risk outcome* describes what will happen if the risk event occurs and before any reactions have taken place. Eg. "non compiling code".
- *Risk reactions* describe what is done after the risk event occurred. There can be more then one reaction; each having different risk effects. Eg. "merge

source code", "revert to last revision"

- The *Risk effect* is the final outcome of the risk event after the reactions have been applied. Eg. "project is late", "overtime required"
- Finally the *utility loss* describes the impact of the risk event on the stakeholders goals, eg. "reduced motivation of the developers".

After the risks have been identified and visualized in risk graphs the items are first prioritized and the the "Risk control planning" is performed. In the order of the priority list all items are then reviewed, and countermeasures are searched to reduce the probability of the risk events and the risk effects.

3.5.2 Step 2: Describe the project organization

Goal: To provide the base for systematically discovering and describing interactions between project members.

Rational: The risk analysis performed in step one provides only a rough overview of the particular communication risks and a more detailed analysis is required to discover concrete interactions that can be automated with notifications. Having a detailed view of the project organization helps to discover potential interaction problems more efficiently. It also helps all project participants to get a shared understanding which tasks are performed by which teams and how the competencies are distributed.

Actors: The project manager has an overview of the project structure; therefore, he or she will draw the "big picture" of it. Team leads (or their proxies) can then extend the view to add team specific details.

Input: The ordered list of risk drivers.

Output: A detailed picture of the project structure from several viewpoints.

Description: We propose an incremental top down approach that draws the socio-

technical system of the project organization from four viewpoints as defined by Wiredu [89]:

1. The *People Viewpoint* describes the organization and distribution of the project teams.

2. The *Information Viewpoint* describes the artifacts (eg. requirements) and their relations.

3. The *Technology Viewpoint* focuses on the technologies that are used in the project.

4. The *Process Viewpoint* relates the elements from the other viewpoints by showing the processes that are performed.

First the viewpoints should visualize the project organization on a rather high level and new details are added in each iteration. There is no standard notation that is suitable for describing all viewpoints; thus, we suggest project managers to use the tools they are already familiar with and to focus on the expressiveness of the diagrams instead of the notation.

driver	**people**	**information**	**technology**	**processes**
dependencies	A	A	C	B
changes	B	A	C	B
geography	A	C	B	A
formality	C	B	C	A
organizations	A	B	A	C
agility	A	C	C	A
culture	A	C	C	A
language	A	B	C	B

Table 3.2: Priorities of the viewpoints depending on the risk driver.

Table 3.2 gives an overview on how important the viewpoints are for particular risk drivers. Project managers should focus on the viewpoints with a priority of A and

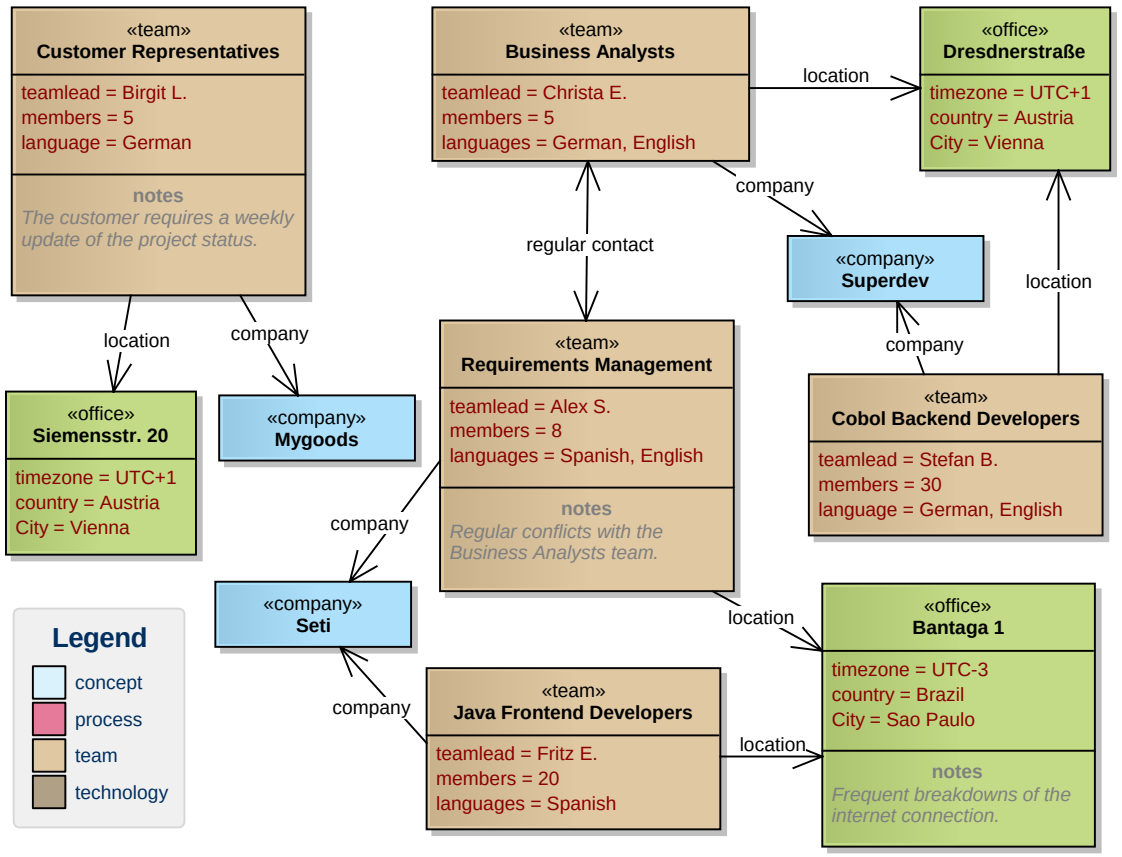

Figure 3.5: Example for the people viewpoint

B and describe them in more detail.

People Viewpoint

The people viewpoints puts all persons that need to communicate with each other in the center of the analysis. Persons are normally grouped into teams that can form a hierarchy and interact with other teams. The viewpoint should also include the offices where the team members are located and the organizations they are working for. Depending on the risk drivers, additional information like the team size, attitudes, timezones, etc. can be included as shown in figure 3.5.

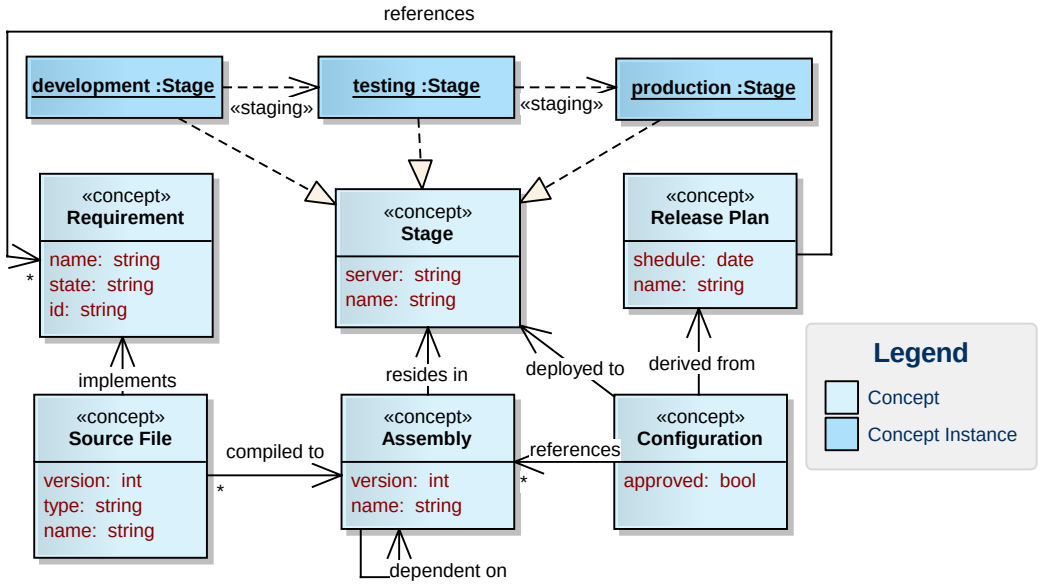

Figure 3.6: Example for the information viewpoint.

Information Viewpoint

Wiredu refers to information "as the lifeblood that circulates between people, processes and technology" [89]. The information viewpoint describes the concepts (eg. requirements, source code, etc.) that are used within the project and their relations. Information about instances of these concepts is communicated between project members (eg. requirement 123 has changed). A detailed view of the concepts and their relations can build a shared vocabulary and provides a reference for new project members. It is suggested to include all information that is commonly used to refer to an instance of a concept (eg. the ID of a requirement). If there is a limited set of instances they can be included as well (eg. stages in the example shown in 3.6).

If project management decides to introduce NOTICON, the information viewpoint can be taken as a reference for creating the context model that is used by the notification rules and in the notification specification language.

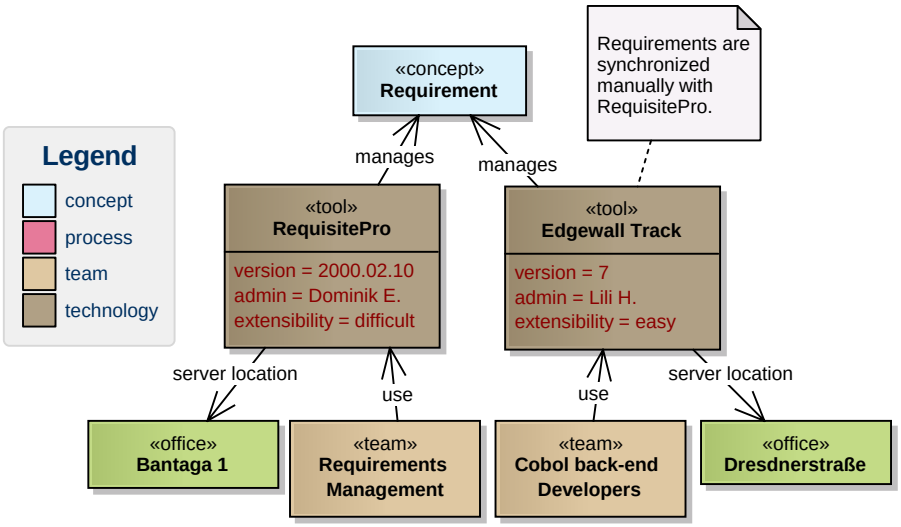

Figure 3.7: Example for the technology viewpoint

Technology Viewpoint

The technology view draws a detailed picture of all technologies (bug tracking systems, email programs, source versioning systems, office suites, etc.) that are used in the project. They should also be linked with the teams that work with them and the information they manage. The view is important to know which applications have to be integrated in a notification solution and which can be extended to sense a user's context.

In the example shown in figure 3.7 we focused on two applications that both manage requirements. Especially in projects where several companies are involved in the development, synchronization between different requirements management tools are common and should be highlighted (eg. with comments).

Processes Viewpoint

The processes viewpoint describes the project based on the activities that are performed. For the analysis of communication it very important because it relates all elements from the three previously described viewpoints. "Processes include all the

driver	focus
dependencies	Focus on processes where elements are changed that have dependencies to other elements.
changes	Focus on processes where the outcome needs to communicated to many teams.
geography	Focus on processes where teams from different locations are involved.
formality	Focus on processes that require frequent communication between teams.
organizations	Focus on processes where teams from different organizations are involved.
agility	Focus on processes that require communication between teams.
culture	Focus on processes that require much communication between teams from different cultures.
language	Focus on processes where teams are involved that have problems to communicate in the project language.

Table 3.3: Relation between risk drivers and the process viewpoint

tasks undertaken by the people such as modeling, programming and testing; all modes of interactions between them, including human-technology interactions; and information generation, processing and transmission tasks." [89]. The granularity of an activity ranges from small tasks to complex business processes.

In a complex software project a whole set of activities need to be performed, but it is not required to describe all of them in the same level of detail. Table 3.3 gives a guideline which processes should be described for which risk driver in detail. Some software development processes like the V Model XT [5] provide a detailed catalog of the activities that are performed within the development process and can be used as starting point.

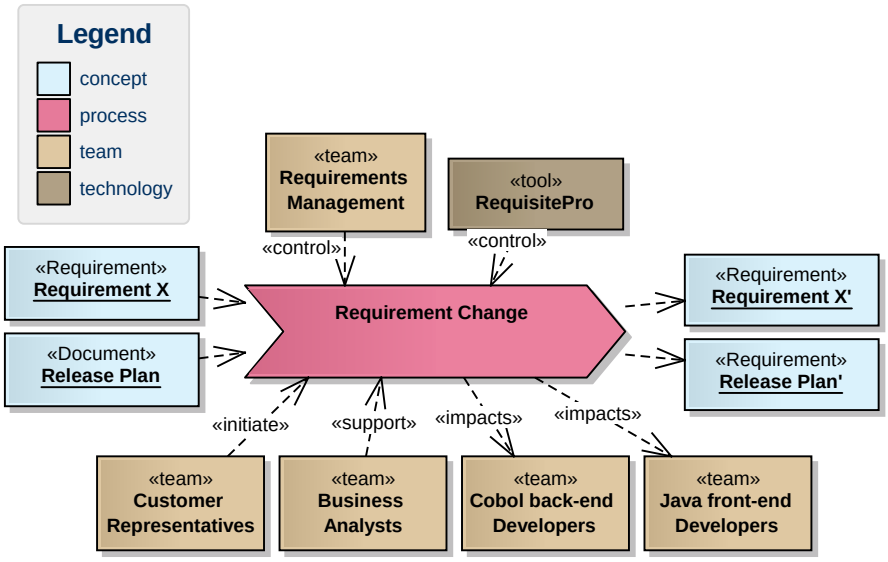

Figure 3.8: Processes Viewpoint

Figure 3.8 shows the "Requirement Change" process. It is initiated by the customer representatives and takes the requirement that should be changed and the release plan as input. Output is a changed requirement and release plan. The process is controlled by the requirements management team and supported by the business analysts team. RequisitePro is the primary tool that is used within the process. We also show the teams that are impacted by the requirement change.

3.5.3 Step 3: Describe the key communications

Goal: To document the key communications that may be automated with a notification solution.

Rational: Loughman et al. [52] argue that a system analysis, as the one we proposed in step 2 (descripe the project organization), is not enough to describe a socio-technical system and find its weak spots: "Despite calls for treating the organization as a total system, systems analysis still focuses more upon processes, data and technology than upon sociotechnology, the fit of technology with its human users." [52]. He argues to extend the system analysis with *communication audits*

that focus on the interaction between people and technology. They are used to analyze both formal and informal information flows, stress the role of culture, and show the clarity, appropriateness, and efficiency of communication.

Actors: The team leads (or their proxies) should describe the key communications which should be reviewed by the project manager.

Input: The description of the project organization from step 2.

Output: An inventory of communication acts that could be automated to mitigate communication risks.

Description: Several instruments, from questionnaires over interviews to content analysis, exist to perform communication audits [36]. De facto standards like the ICA communication audit [32] provide project managers with a well validated toolbox. If the communication risks that were discovered in step 1 are very high and the project is reasonably large, a full communication audit may be beneficial. Otherwise we propose a lightweight approach that builds on the idea of Dietz [18] and Geurts [30]:

Communication between two parties (human or machine) is compromised by a series of communication acts. For each communication act we can describe the the *performer*, the *receiver*, the *intention*, the *message*, the *time-for-completion*, the *channel*, and the *trigger*:

The *performer* is the person or machine who initiates the communication act. The *receiver* is the person or machine who receives the message. The *intention* specifies the goal of the communication act. The *message* conveys the information that gets transmitted. The *time-for-completion* is the time by which the performer expects a reaction of the receiver. The *trigger* describes what business events trigger the communication act, and the *channel* states how the message is transmitted. An example of a communication act is shown in listing 3.1.

```
performer: Business Analysts
receiver: Requirements Management
intention: directive to perform an impact analysis
message: Client requests a change of requirement X
time-for-completion: three days
```

```
6 medium: phone
7 trigger: Client informed Business Analysts of a changed business functionality
```

Listing 3.1: Example for a communication act in the "Change Requirement" process shown in figure 3.8

For each business process defined in step 2 the key communications should be described according to the format defined above. It is not necessary to describe all communication acts in this level of detail; focus should be be put on [85] communication acts with 1) a frequent occurrence (eg. hourly), 2) high risks (depending on the risk drivers), and 3) that are significantly important to support collaboration.

3.5.4 Step 4: Define the Notifications

Goal: To provide an inventory of all notifications that can be implemented with notification solutions like NOTICON.

Rational: A formal definition and inventory of the notifications provides users with a single point of truth regarding everything that concerns notifications. NOTICON can interpret the notification definition and create notifications accordingly.

Actors: Project management and team leads will define the business part and administrators the technical part of the notification definition.

Input: The communication acts described in step 3.

Output: An inventory of notification definitions.

Description: We divided a notification definition into a business part and a technical part: The business part describes the notification from a business point of view and elaborates on the rationals why the notification should be introduced. The technical part contains the formal definition of the notification (the notification specification) and implementation issues. The structure is based on the one defined by Fabian et al. [21], but has been extended slightly.

The business part consists of:

- **Title:** An expressive title to reference the notification more easily.
- **Description:** The description of the notification in an informal way.
- **Scenario:** A little scenario should highlight the benefits of the notification and provide users with a quick way to judge if the notification is relevant for them.
- **Upsides:** The upsides should describe all benefits that are expected of the notification.
- **Downsides:** The downsides describe all negative effects that the notification can cause. Especially the impact of false positives and false negatives should be described.

The technical part consists of:

- **Notification Specification:** The notification specification is the most important part of the notification definition because it defines the notification in a formal way. We propose a domain specific language that is easy to understand by business users but expressive enough to specify complex notification scenarios. It is described in detail in section 4.5.
- **Implementation Issues:** Describe which information from which systems have to be gathered to be able to create the notifications.

Example for a Notification Definition

Title: Requirement Changes

Description: If a requirements manager changes a requirement that is currently used by a developer, the developer will be informed in his/her development environment immediately.

Scenario: The requirements manager Alex gets informed by the customer that the formula for calculating discounts has changed, thus changes the requirement

accordingly in RequisitePro. Birgit who is currently implementing the discount functionality with Eclipse gets immediately information about the change in the Eclipse notification panel.

Upsides: No time is spent working on old requirements.

Downsides: A trace between requirements and source code needs to be maintained.

Notification Specification:

```
1 Title: Requirement Changes
2 Receiver: any User ?user
3 Context:
4 the ?user "uses" a Requirement ?req and
5 the ?user "uses" the Tool "Eclipse" and
6 the description of ?req changes
7 Deliver: immediate
8 Channel: "Eclipse"
9 Subject: "Requirement {?req.name} has changed"
```

Implementation Issues: – RequistePro has to send an event to the notification server when a requirement changes. – Eclipse has to be extend with a sensor that gathers the requirements a user currently works on. – Eclipse has to be extended with a publisher that displays notifications to the user.

4 Prototype Development

This chapter provides a detailed overview of the architecture of NOTICON and the notification specification language.

NOTICON is a context-aware notification system written in Java. The core component is the Drools rule engine that manages the context model and the rules, which trigger the notifications. For application integration and message routing we use the enterprise service bus "Mule" [1], and Apache ActiveMQ [2] for communication from and to the service bus.

The components NOTICON is built of and their interactions are explained in section 4.1. A message model defines the messages that are routed within the service bus and is pictured in section 4.2. Section 4.3 describes the context model that is used by the rule engine. An agenda controls the order on which a set of rules get executed by the rule engine and is described in section 4.4. Finally we explain the notification specification language in section 4.5 that is used by project members to specify the notifications.

4.1 Components

The architecture of NOTICON is event driven; thus, it is characterized by a very loose coupling between its components. An enterprise service bus manages the routing of events between the components. We decided to use the ESB Mule because of its highly scalable SEDA processing model (see [87]), its wide use in industry, and its

[1] http://mule.codehaus.org
[2] http://activemq.apache.org

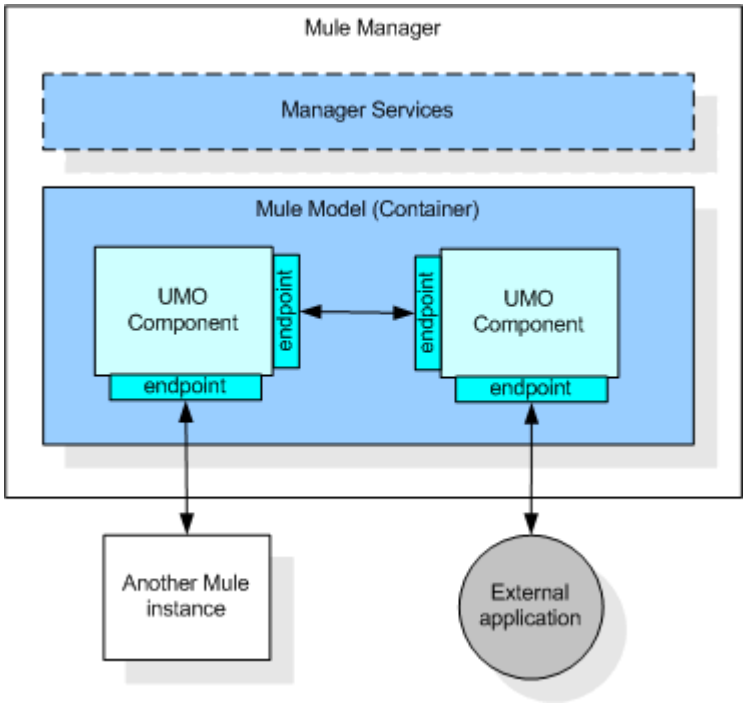

Figure 4.1: Mule Server Components [6]

permissive Apache license.

Figure 4.1 illustrates the components of Mule: The *Mule Manager* manages the bootstraping of the service bus and its internal components. The *Model* is a container in which the user components reside. It provides services like transaction management, event routing, or logging and is responsible for threading, pooling, and life cycle handling of the hosted components. *UMO* stands for Universal Message Object and is a POJO (plain old Java object) that execute logic on incoming events and returns new events as output. Typically, UMOs contain the business functionality that should be executed on the arrival of events. They communicate with each other via *endpoints* that define a communication channel between two or more UMOs. A channel can be configured with filters, transformers, and interceptors to control exactly which events are routed through.

Figure 4.2 illustrates the core components of NOTICON . With the exception of the Rule Engine and the Notification Management components there are typically several instances of a component. We describe the components in the order that

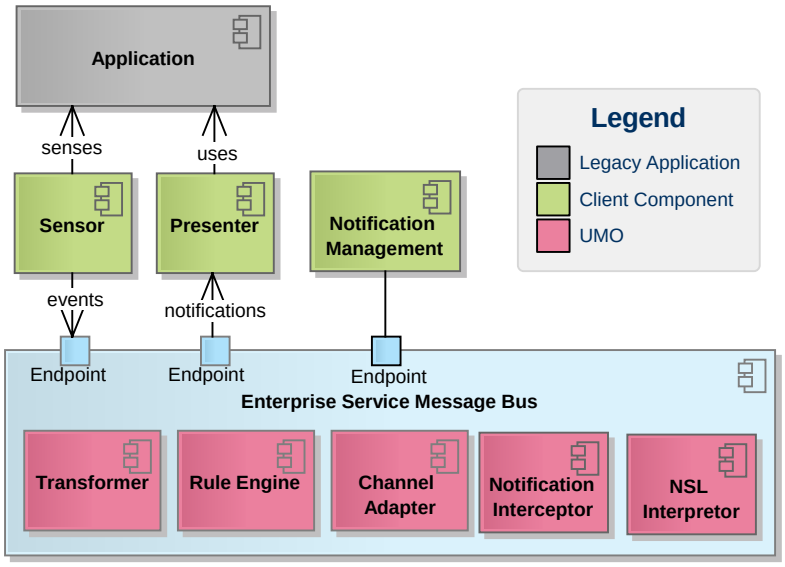

Figure 4.2: Core components of NOTICON

events typically flow through the system:

- *Sensors:* Sensors reside on the client side and are responsible for monitoring the project context (see 2.2.2 on page 20) and sending context information to endpoints of the service bus. Mule has a wide range of built-in endpoint types (eg. JMS, WebService, Email, etc.) which eases the development of sensors.

- *Transformers:* Transformers transform messages sent by the sensors into elements of the context model (see section 4.3). They wrap those elements into command messages [41] that the Rule Engine component can interpret. Eg. an Email with the subject "Requirement X has changed" that is sent to an email endpoint, would be translated into an `AddOrModify` message that contains an object of the type `Requirement`.

- *Rule Engine*: The Rule Engine component is the central component of NOTICON. It wraps an instance of the the Drools rule engine [3] that is responsible for reasoning over the context model and the creation of the notifications. Depending on the incoming events, the Rule Engine component retracts facts from the rule engines working memory or asserts new ones. It also checks if

[3] http://labs.jboss.com/drools/

objects contained in an incoming message already exist in the working memory and merges them accordingly.

- *Channel Adapters:* Users can define the channel that should be used to transmit the notification to the receivers in the notification specification (ec. via Eclipse). For each channel there exists one Channel Adapter component that transforms the internal notification representation into a message that gets either directly transmitted to the receiver (eg. by Email) or is dispatched to a Presenter that then displays the notification to the user (eg. a Presenter could display notifications in an Eclipse panel).

  The routing of the notifications from the Rule Engine component to the Channel Adapter components is configured in the Mule Model. This enables scenarios where notifications should be sent to some channels regardless of the notification specification (eg. a personalized RSS feed, or a channel that persists all notifications).

- *Notification Interceptors:* Notification Interceptors allow additional processing of notifications before they are delivered to presenters. They can perform functionality like notification encryption, calculation of metrics, filtering based on user preferences, or persisting all notifications to a database.

- *Presenters:* Presenters are components that are installed on the client machines and responsible for displaying notifications to users. They typically extend an application like Eclipse or MS Word and display notifications within that application. Integrating notifications into existing tools can significantly reduce the interruption of the user's primary activities because no context switching is required. Presenters are typically connected to the service bus via ActiveMQ, although Channel Adapters may implement different strategies to inform the presenters about new notifications.

  The final destination of a notification does not necessarily have to be a user, but could also be an application. Eg. a notification that is sent whenever the users closes Eclipse could automatically trigger a SVN commit of the users working copy. The publisher ultimately decides what to do with the notification and

controls the operations that are performed on receipt.

- *Notification Management:* The Notification Management component implements the client interface for NOTICON. It provides functionality to 1) browse all notification specifications, 2) create new notification specifications, and 3) subscribe or unsubscribe from notifications. In NOTICON we implemented the Notification Management component as a simple web application deployed on a Tomcat server.

- *NSL Interpretor:* The NSL Interpretor component compiles notification specifications that are defined with the Notification Management component into the Drools rules format. The rules are wrapped into `AddRule` messages and routed to the Rule Engine component.

4.2 Message Model

The message model describes the messages that are routed between the components. A message represents an event and a component can be both event consumer and event producer. Routing rules are defined in the Mule configuration and describe which events are routed to which components under which conditions. Advanced routing rules can be specified, that eg. inspect the message content and route the event accordingly.

```
1  <mule-descriptor name="NotificationToChannelBridge"
2                   implementation="org.mule.components.simple.BridgeComponent">
3   <inbound-router>
4     <endpoint address="vm://notifications" \>
5   <\inbound-router>
6
7   <outbound-router>
8     <router className="FilteringOutboundRouter">
9       <endpoint address="vm://email.channel"/>
10        <filter expression="channel='email'"
11                className="OGNLFilter"/>
12        </filter>
13    </router>
14    <router className="FilteringOutboundRouter">
15      <endpoint address="vm://eclipse.channel"/>
```

```
16        <filter expression="channel='eclipse'"
17          className="OGNLFilter"/>
18        </filter>
19    </router>
20  </outbound-router>
21 </mule-descriptor>
```

Listing 4.1: Example for a routing configuration

Listing 4.1 shows the configuration of how notification messages are routed from the "vm://notifications" endpoint to the endpoints of the Channel Adapters. Depending on the `channel` property of the `Notification` object, the notification is routed either to the "vm://email.channel" endpoint or to the "vm://eclipse.channel" endpoint.

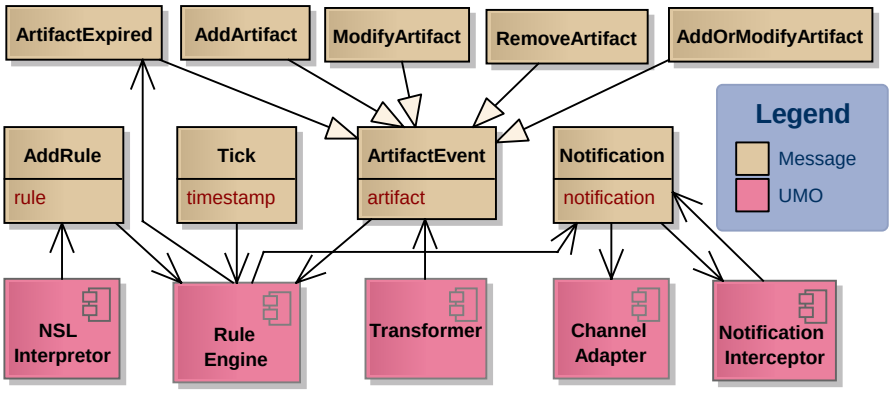

Figure 4.3: Message model of NOTICON that shows the messages and the components that consume or produce them.

Figure 4.3 illustrates the most important message types of NOTICON and the components that produce or consume instances of these types:

- *AddRule:* An `AddRule` message is generated by the NSL Interpretor whenever the user creates a new notification definition. It contains a string representation of the rule in the Drools specific rule format.

- *Tick:* `Tick` messages are sent by a special endpoint that creates instances of this type at a predefined interval. On arrival of a `Tick` event, the Rule Engine components modifies the `timestamp` property of the `PseudoClock` artifact in

the rule engines working memory, which provides the base for time sensitive notifications.

- *ArtifactEvent:* An `ArtifactEvent` message contains an object of type `Artifact` (see 4.3) and specifies the action that should either happen with the artifact, or has happened with it. The Rule Engine component modifies the working memory of the rule engine depending on the subtype of the `ArtifactEvent` message:

  - On `AddArtifact` messages, it adds the artifact into the working memory and overwrites any existing artifacts that have the same ID.
  - On `ModifyArtifact` messages, it retrieves the artifact with the same ID from the working memory and merges the two objects. If no artifact with the same ID exists in the working memory, the message is discarded.
  - On `AddOrModifyArtifact` messages the artifact is added if it does not exist in the working memory or updated if it does.
  - On `RemoveArtifact` messages, the artifact is retracted from the working memory.

  `ArtifactExpired` messages are generated by the Rule Engine component when an artifact has expired and got retracted from the working memory.

- *Notification:* `Notification` messages are generated by the Drools engine according to notification specifications.

4.3 Context Model

The context model defines the types of the objects that can be inserted into the rule engines working memory. A type is a Java class and an object is an instance of a class. *Transformer* components translate objects of other types (eg. a legacy email message object) into objects of types defined in the context model.

We defined a type hierarchy consisting of three layers:

1. The *core artifacts layer* defines the root of a type hierarchy and general concepts that are needed for processing the rules. Only objects of these types (or deduced types) can be processed by the Rule Engine component.

2. The *common artifacts layer* predefines types that are commonly used concepts in GSD projects like "User" and "Location".

3. The *user artifact layer* contains types that inherit from the upper layer and is usually created specifically for a project by an administrator. The types defined in this layer usually reflect the concepts and relations that were defined in the *Information Viewpoint* in step 2 of our process (see section 3.5.2 on page 3.5.2).

The next sections will describe the layers in detail.

4.3.1 Core Artifacts Layer

The core artifact layer contains predefined types that are required to fulfill the functionality of NOTICON (see figure 4.4):

- *Artifact:* Any type of the context model has to inherit from the `Artifact` class or one of its subclasses. We refer to all objects of this type as "artifacts". An `Artifact` has the following properties:

  - The `ID` is a string that is used to reference an artifact. It does NOT have to be unique although it is suggested to be unique for all objects in the GSD or user layer. It should have a meaningful value for users (eg. "Benedikt Eckhard") because artifacts can be referenced by ID in the notification specification language (see section 4.5).

  - Artifacts are not kept endless in the working memory, but expire after a timespan that can be defined by the administrator. The `ExpirationDate` contains a timestamp when the artifact should expire and be retracted

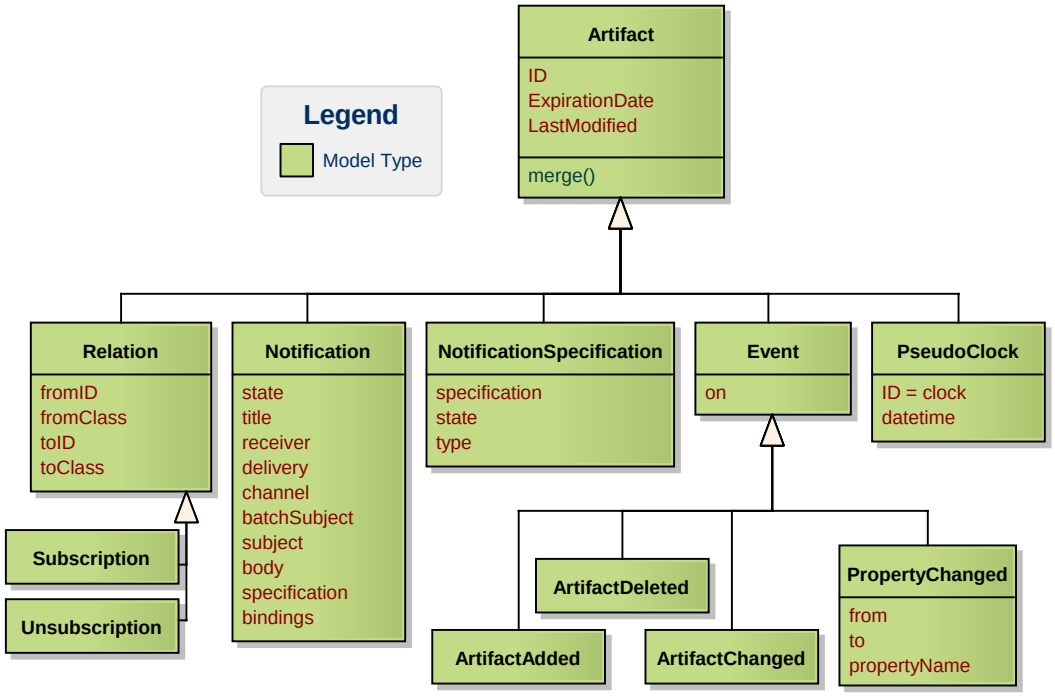

Figure 4.4: Core artifacts layer

from the working memory. Then, an `ArtifactExpired` event is generated which Interceptor components can react upon.

– The `lastModified` property contains a timestamp when the artifact was last modified.

Types that inherit from Artifact must implement the `merge()` function. An artifact thats `merge()` method gets called must update its values with that of the artifact that was passed as the parameter. The function returns a `PropertyChanged` artifact for each property that has changed as illustrated in figure 4.5. The merge functionality is very important, because it generates the events that users can use in notification specifications.

- *Relation:* Artifacts of type `Relation` establish a directed relation between two artifacts. The ID of the artifact is the relation name (eg. "uses") and the properties (`fromID, fromClass, toID, toClass`) specify exactly between which artifacts it is established.

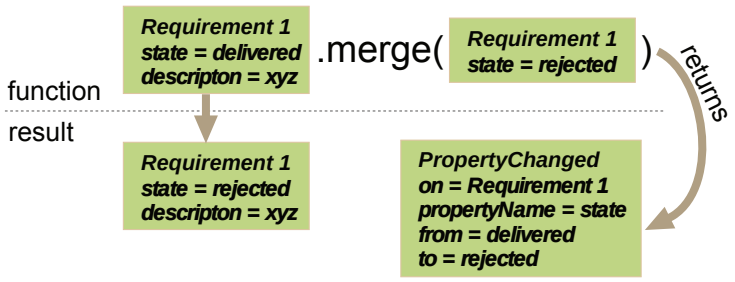

Figure 4.5: Illustration of the merge function

Relations are critical for complex notification specifications and can be sensed and created at any layer of the architecture. Eg. an Eclipse sensor could parse the actual source file and check if a trace to a requirement is established via a comment; it could then send this information to the service bus. Another scenario is that there is a special sensor for relations that uses tools like JDepend to analyze components and then publish the discovered relation information to the rule engine. It is also possible to define rules that automatically establish relations between artifacts when some conditions become true; eg. for making transitive relations explicit. There is no general guidance on how relations should be created, and it depends very much on the concrete scenario.

There are two subclasses of `Relation` that both relate an artifact of type `Artifact` with an artifact of type `NotificationSpecification`. A `Subscription` artifact defines that the `from` artifact can be a receiver of notifications created by this specification; `Unsubscription` artifacts define the opposite.

- *NotificationSpecification:* For each notification specification defined by users, an artifact of type `NotificationSpecification` is inserted into the working memory of the rule engine. This is needed to give users the ability subscribe to or unsubscribe from notifications. The text of the notification specification is stored in the `specification` property. The `state` defines, whether a notification specification is still active - a deletion of rules from the rule base would require a restart of the server; thus, they are only deactivated if they are not needed any more. The `type` property specifies whether users have to subscribe to receive notification from this specification or have to unsubscribe

if they do not want to receive notifications.

- *Notification:* When the rules for creating a notification match, an artifact of type `Notification` is added to the working memory with its `state` property set to "undelivered". If all conditions for sending the notification are met, the `state` property gets changed to "ready for delivery". The Rule Engine component picks all rules that are "ready for delivery", wraps them in a `Notification` message, and dispatches it to the service bus where it is then routed to the `Channel Adapter` components. After that it changes the state to "delivered" or "error" if there were some errors.

  The properties `receiver`, `delivery`, `channel`, `batchSubject`, `subject`, and `body` are described in section 4.5 in detail.

  The *specification* property contains the ID of the notification specification that created the notification. The *bindings* property contains all variables that the user used in the notification specification and the objects that they are bound to. What bindings are, is also described in section 4.5.

- *Event:* Artifacts of type `Event` allow users to define rules that react on changes in the context model, eg. when the state of a requirement changed to a defined value. The Rule Engine component is responsible for creating the `Event` artifacts when it changes something in the working memory. They are discarded immediately after all rules had the chance to react on them. The *on* property contains the artifact that the event is related to. `PropertyChanged` artifacts do not relate to an artifact as a whole, but just to a property value of an artifact. They contain the value that an artifact had previously (in the `from` property) and that it has now (in the `to` property). The name of the property the change relates to, is stored in the `propertyName` property. `PropertyChanged` artifacts are generated by artifacts themselves when their `merge` function is called.

- *PseudeoClock:* At startup of the server, one artifact of type `PseudoClock` with the ID "clock" is added to the working memory. It contains the current date and time in the `datetime` property which get updated on every `Tick` message.

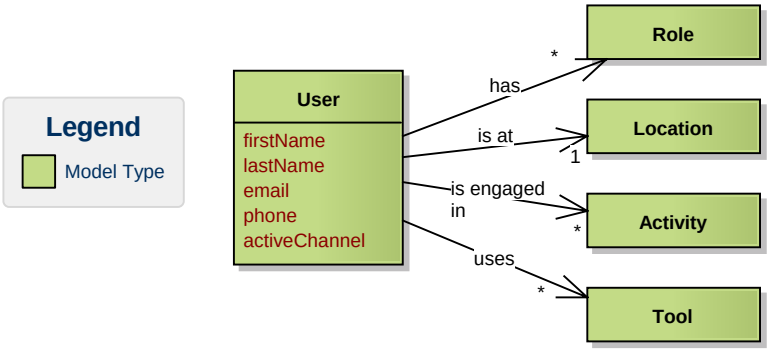

Figure 4.6: Common Artifacts Layer

The pseudo clock is needed to check if artifacts are expired and should be retracted from the working memory.

4.3.2 Common Artifacts Layer

This layer contains a number of artifact types that are very commonly used in GSD projects (see figure 4.6). They should serve as a starting point for administrators which can subclass from these common artifacts or add predefined instances. The arrows between the types define the IDs of relation artifacts that connect two artifacts of these types.

Example: If there is an artifact of type `Activity` with the ID "implementing requirements" and a user is engaged in this activity, an artifact of type `Relation` with the ID set to "does" would be created that connects the user and the activity artifact. A notification specification can use this information eg. to send notifications only to users that are currently engaged in that activity as shown below:

```
1 Receiver: any User ?user
2 Context:
3 the ?user "is engaged in" the Activity "implementing requirements"
```

We predefined the following types:

- *User*: There is typically one instance of this type for each project member in the GSD project. Although not enforced, user artifacts (representing the real

persons) are usually the receiver of notifications. The `activeChannel` property can be set to the notification channel that is currently the least interruptive for the user. Eg. if the user Philip is currently working in Eclipse and Eclipse can receive NOTICON notifications, then the active channel property would be set to "Eclipse".

- *Role*: Instances of this type could be "Project Manager", "Developer", "Team Lead", etc. Users can have more then one role.
- *Location*: `Location` artifacts represent locations like "office in Vienna". At a point in time a user can only be at one location; we predefined a rule that enforces this constraint.
- *Activity*: Users can be engaged in more then one activity concurrently.
- *Tool*: Instances of this type could be eg. "Eclipse", "MS Word", etc. Depending on the tool a user is currently using, an appropriate channel could be selected in a notification rule.

4.3.3 User Artifact Layer

The user artifact layer contains all subclasses of `Artifact` that have been created by the administrator specifically for a project. Typically the types reflect the concepts in the information viewpoint and should be well understood by all users that specify notifications. Currently it is not possible to add new artifact types at runtime; a restart of the server would be required. The current state of the working memory would be persisted and restored after startup. New relations can also be defined during runtime.

4.4 Rule Engine Agenda

The rule engine agenda controls the order in which rules are processed after changes in the working memory. It should be understood by administrators who want to

add new legacy rules (written in the Drools rule format).

On arrival of an `ArtifactEvent` message, the Rule Engine component executes the following process:

1. The artifacts contained in the message are merged with the artifacts that already exist in the working memory. The `Event` artifacts that were generated during the merge operation are added to the the working memory as well.

2. Then the rule-set that controls the aging of artifacts is evaluated. It checks all artifacts if their `expirationDate` property is smaller then that of the `PseudoClock` and removes them accordingly. On removal, `ArtifactExpired` events are generated and sent to a predefined endpoint.

3. The next rule-set that is executed, contains rules that were defined by administrators and usually enforce constraints (eg. a user may only be at one location at the same time) or create deductions (eg. if the component A depends on B and B depends on C, then A indirectly depends on C).

4. In the next step the rule-set that creates the notifications according to the notification specifications is evaluated. If the conditions for the delivery of a notification are met, the `Notification` artifact is wrapped in a `Notification` message and sent to a predefined endpoint.

5. In the last step, all artifacts of type `Event` are retracted from the working memory.

4.5 Notification Specification Language

The notification specification language (NSL) is a domain specific language [51] that allows users to formally specify their notification requirements. It shields users from the underlying complexity of the rule engine and separates the business perspective from the technical implementation. This gains the benefit that the technical implementation can be changed without affecting existing specifications and that

multiple languages could be supported by the same implementation. The specification language could also be implemented by other product vendors to provide a uniform experience for users, regardless of the tools they are using.

Our design goal for the language were:

- It must be easy to understand and to write by business users who have no prior knowledge in programming languages and/or rule engines.
- It must be expressive enough to support a wide variety of notification scenarios.
- It can be easily compiled into the Drools specific rule format.

4.5.1 Notation and General Concepts

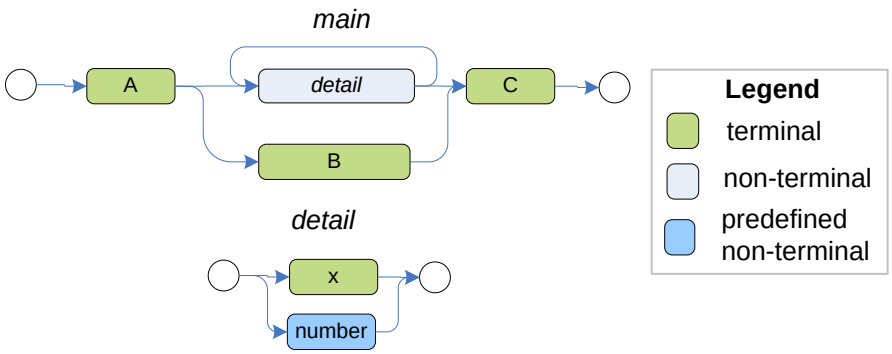

Figure 4.7: Simple grammar described in a graphical notation.

We decided to describe the grammar of the NSL in a simple graphical notation as shown in figure 4.7. A grammar consists of productions that define a sequence of symbols that they can produce. Symbols are either terminals or non-terminals. Terminals become part of the string that is generated by the production; thus, they are not further replaced. Non-terminal point to another production that produces the content that replaces the non-terminal. A string is valid according to a grammar if it can be produced by the productions.

The example in figure 4.7 defines a very simple grammar with two productions: The *main* production can produce a string that starts with the letter A and ends with

symbol	explanation	example
ID	a valid Java identifier (see [33])	benediktEckhard
qstring	a string enclosed in quotation marks	”My String”
datetime	a date and/or time specification enclosed in # signs	#30.04.1982#
number	a float or an integer	13.2
boolean	either ”true” or ”false”	true
model type	refers to any type that has been defined in the context model	User

Table 4.1: Predefined productions

the letter C. In between can either be the letter B or any number of strings that are produced by the ”detail” production. The detail production can either produce the letter ”x” or a number (eg. 240 or 92.007). Strings that can be validated by the grammar would be eg.: ”ABC”, ”AxC”, ”AxxC”, ”A240C”, ”A92.007x240C”, etc.

The arrows between the symbols point to the next symbol the production can produce. If an error points to more than one symbol, the production can produce all of them. Green shapes contain string terminals. Light blue shapes point to other productions and dark blue shapes point to commonly used productions that we defined in advance and which are explained in table 4.1. An addition production has been defined that is not visualized in the diagrams: Between two symbols (terminals, or non-terminals) at least on blank is produced.

Figure 4.8 shows the main production rule for the notification specification language. Any specifications a user creates is validated with this production. The specification starts with a *title* and is followed by the specification of the *receivers* and the *context* that define when the notification should be created. Optionally conditions can be specified that delay the *delivery* of the notification. The *channel* over which the notification should be sent is specified next (eg. Eclipse). If the user defined that the notification should be sent as a batch mail the *batch subject* needs to be set. Compulsory for each notification is a *subject* and optionally a message *body* can be set.

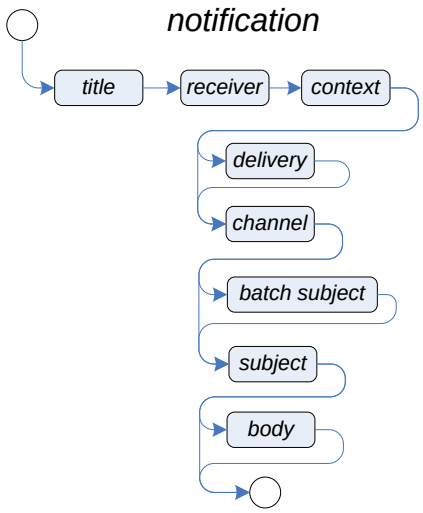

Figure 4.8: Main production rule

The next sections will describe the elements in detail:

4.5.2 Title

Each notification specification has to have a title that is unique within the system. The title is set on each notification object that is created by the rule specification and can be used eg. in the routing rules. Users also need to know the title if they want to subscribe to or unsubscribe from notifications sent by the specification.

Figure 4.9: Notification title

The title is specified according to the production shown in figure 4.9. It starts with the string "Title:" followed by a quoted string that contains the title and a newline character. An example with the the title of the notification set to "This is my notification" is given below:

```
1 Title: "This is my notification"
```

4.5.3 Receiver

For each notification a receiver has to be specified. Any object that exists in the working memory of the rule engine can be receiver of notifications - regardless of its object type. It is therefore perfectly legal to define a notification with an artifact of the type "Situation" as the receiver. It is up to the channel adapters to decide what to do with the notifications. Typically a notification specification is not defined for a single receiver but a group of them (eg. all users that have brown hair). The receiver can decide if he or she wants to receive notification; whether a receiver has to subscribe or unsubscribe can be defined by the creator of the specification.

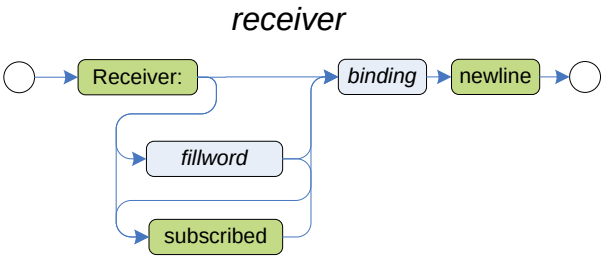

Figure 4.10: Notification receiver

Figure 4.10 shows how receivers are defined. The definition starts with the string "Receivers:" and is followed by a binding that specifies the artifacts that should receive the notification. Optionally the binding can be prefixed with the keyword "subscribed" that specifies that receivers have to subscribe first to receive the notifications. For better readability a *fillword* (eg. "any") can follow the "Receiver:" keyword.

```
1 Receiver: the User "Benedikt"
```

The above example specifies that the user "Benedikt" should receive the notifications. Or more specifically: it adds a proposition to the notification rule that there has to exist an artifact in the working memory that is of type *User* and its ID property set to "Benedikt". If there is only one artifact that matches the condition only one notification would be sent.

A binding binds artifacts to variables. Figure 4.11 shows that a binding always

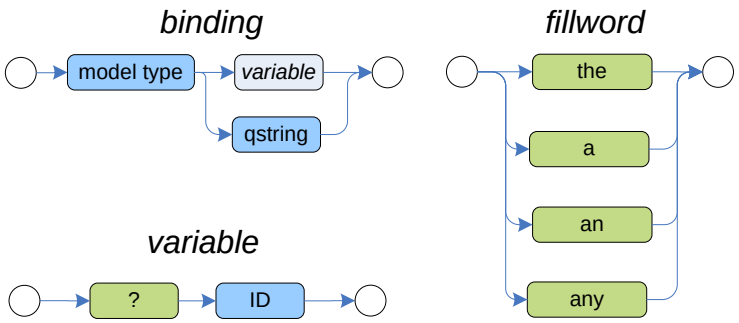

Figure 4.11: bindings, fillwords and variables

consists of a model type (eg. *User*) and either a quoted string (eg. "Benedikt") or a variable. The example below shows the use of variables in bindings:

```
1 Receiver: any subscribed User ?user
```

The example above reads that the notifications should be sent to all users that subscribed to the notification. To be able to further constrain which user will receive the notification, any artifact of type `User` is bound to the variable named "?user". Subsequent statements that reference the "?user" variable would be executed once for each user and with the "?user" variable bound to a concrete object (eg. User "Benedikt"). The semantics of variables are similar to "foreach" loops in programming languages as illustrated in the following Java example:

```
1 foreach(User user : users) {
2   if(user.isSubscribedTo("specification")) {
3      sendNotificationTo(user);
4   }
5 }
```

Variables in the notification specification always have to start with a "?" character followed by a string that is a valid Java identifier. This basically restricts variable names to start with a character and not to contain any whitespace or special characters like ".".

We found out that users can better write and read notification specifications if they can use words like "the","any", etc.; thus, we defined a list of words that can be used but have no meaning for the implementation. "Receiver: the User ?user" has exactly the same meaning than "Receiver: any User ?user".

4.5.4 Context:

In the context part conditions can specify on a very fine granularity when a notification should be generated. Also the receivers can be further restricted. The context definition starts with the string "Context:". Subsequent lines specify conditions until no "and" string prefixes the newline character.

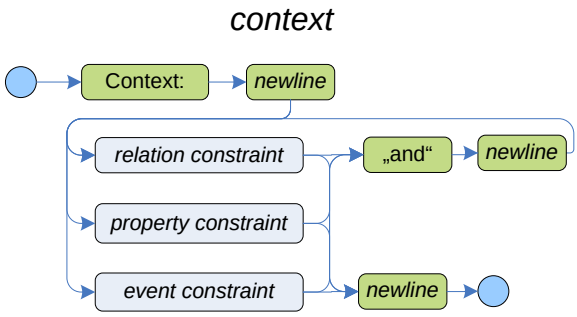

Figure 4.12: Context definition

Three types of constraints are possible as shown in figure 4.12:

- *relation constraints* specify that there has to exist a relation between artifacts in the working memory that matches the pattern.
- *property constraints* constrain the value of properties of artifacts.
- *event constraints* wait for events to happen on artifacts or their properties.

Relation constraints

As described in section 4.3 on page 83, artifacts can be related to each other with artifacts of type `Relation`. *Relation constraints* search for `Relation` artifacts in the working memory that have 1) the ID set to *relation name*, 2) an artifact that matches the left pattern in the "from" property, and 3) an artifact that matches the right pattern in the "to" property. The left and the right pattern refer to the binding or variable on the left or right hand side of the *relation name*.

```
1 Context:
```

```
2 the User "Benedikt" "has" the Role "Developer"
```

The example above checks if Benedikt is a developer by searching for a relation artifact with the name "has" that has the user "Benedikt" in its `from` property and the role "Developer" in its `to` property.

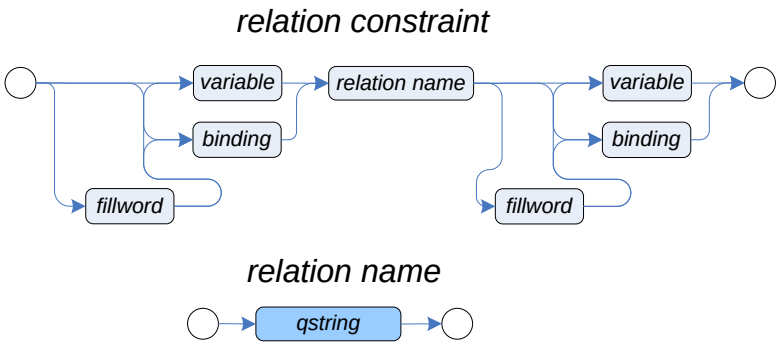

Figure 4.13: Relation constraint

As shown in figure 4.13 on the left and right hand side of the relation name can be either a binding or a variable. Just the variable can only be used if it has been bound previously to an artifact as shown in the following example:

```
1 Receiver: any User ?user
2 Context:
3 the ?user "has" the Role "Developer"
```

The variable "?user" has been used before which ensures that is bound to a concrete artifact when the relation constraint is checked. With bindings new variables can be introduced as shown below:

```
1 Context:
2 the User "Benedikt" "uses" a Tool ?tool and
3 the User "Birgit" "uses" the ?tool
```

This searches for "uses" relations between the user "Benedikt" and artifacts of type `Tool` which are bound to the variable `?tool`. In line 3 relations between the user "Birgit" and artifacts bound to the ?tool variable are searched. The context would match for any tool that both "Benedikt" and "Birgit" use.

Property constraints

Property constraints allow to constrain the property values of artifacts. The following example constrains the receivers of the notifications to all users that have "Benedikt" as their first name:

```
1 Receiver: User ?user
2 Context:
3 the firstName of the ?user is "Benedikt"
```

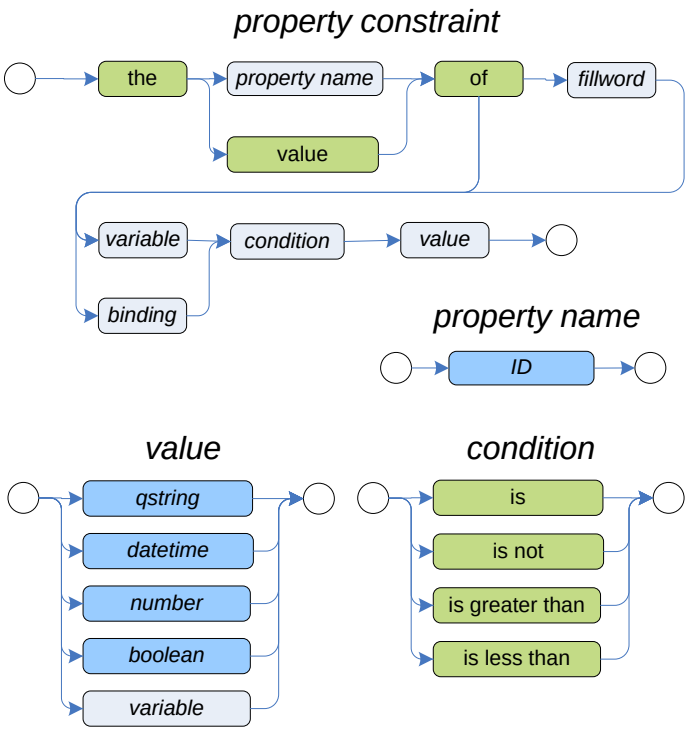

Figure 4.14: Property constraint

As shown in figure 4.14, a property constraint starts with the string "the" and is followed by either the property name or the string "value". Then follows the string "of" and the artifact thats property value should be checked. After that comes the condition that should be checked. Users can use variables on the right hand side and on the the left hand side of the condition. Example:

```
1 Receiver: User ?user
2 Context:
3 the name of the Tool ?tool is not "Eclipse"
```

This would bind the variable `?tool` to all tools that do not have the name "Eclipse". As in the *relation constraint*, unbound variables have to be prefixed with the type of the artifact, unless they occur on the right hand side of the condition as shown in the following example:

```
1 Receiver: User ?user
2 Context:
3 the firstName of the ?user is ?name and
4 the lastName of the User ?other is ?name
```

The statement in line 3 binds the `firstName` property of the artifact that is bound to the `?user` variable to the unbound `?name` variable. In the following statement the `lastName` properties of all users are compared to the value bound to the `?name` variable. If the `lastName` property of a user matches the value bound to the `?name` variable (the first name of a receiver), the user artifact is bound to the `?other` variable. The notification would be sent once for each user that has a first name that equals the last name of another user.

It is sometimes required not to compare a property value of an object, but to compare values directly. This can be done by using the keyword "value" instead of a property name as shown below:

```
1 Receiver: User ?user
2 Context:
3 the birthDate of the ?user is ?birthdate and
4 the value of ?birthdate is less than #30.04.1982#
```

In line 3 the value of the `birthDate` property is bound to the `?birthdate` variable. In line 4 the value of the variable is checked whether it is before a specific date.

Currently we only support four types of conditions (is, is not, is greater than, is less than), but we plan to extend the list in future versions.

Event constraints

When an artifact in the context model changes, the *Rule Engine* component adds special event artifacts to the working memory that further describe the changes.

Event constraints use these artifacts to provide users with a simple notations to describe the events that should trigger a notification.

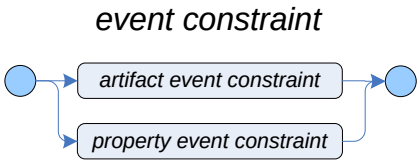

Figure 4.15: Event constraint

Figure 4.15 shows that an event constraint can either be an *artifact event constraint* or a *property event constraint*. Artifact event constraints watch for events that relate to an artifact as a whole, while property event constraints can react on property changes of artifacts.

```
1 Receiver: any User ?user
2 Context:
3 the ?user "uses" the Requirement ?req
4 ?req changes
```

The example shown above would create notifications whenever a requirement changes, but only for users that are related to the changing requirement with a ”uses” relation.

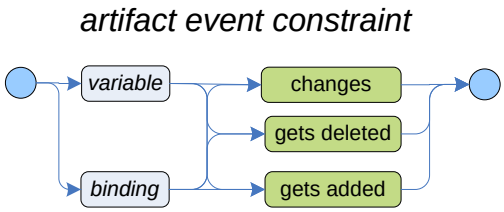

Figure 4.16: Artifact event constraint

Figure 4.16 shows that users can react on changes, deletions, and additions of artifacts. As with relation and property constraints, both bound variables and bindings can be used to specify the artifact that the event must be related to.

With *property event constraints* users can react on property changes of artifacts. First the property that the constraint relates to is defined, followed by the string ”changes” (see figure 4.17). Optionally users can further constrain the event by

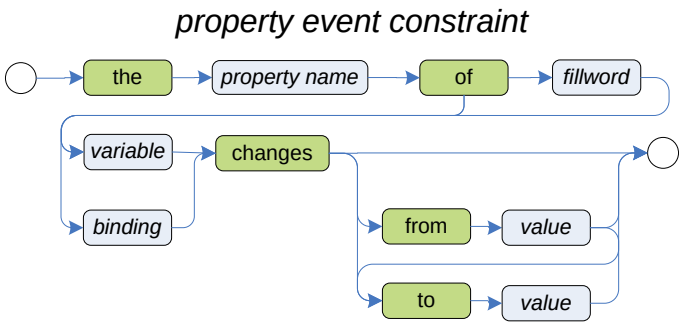

Figure 4.17: Property event constraint

specifying how the value of the property was before and after the change as shown in the following example:

```
1 Context:
2 the state of the Requirement ?req changed from ?x to "rejected"
3 the value of ?x is not "deployed"
```

The conditions would match if a requirement changes to the state "rejected", but only if the previous state was not "deployed". As shown in figure 4.17 both the *from* and the *to* values are optional. Only the order of the two has to be maintained.

4.5.5 Delivery options

Users have the possibility to define when the notifications produced by a specification should be delivered. If no option is specified, the notification is delivered as soon as all conditions that were defined in the *context* match.

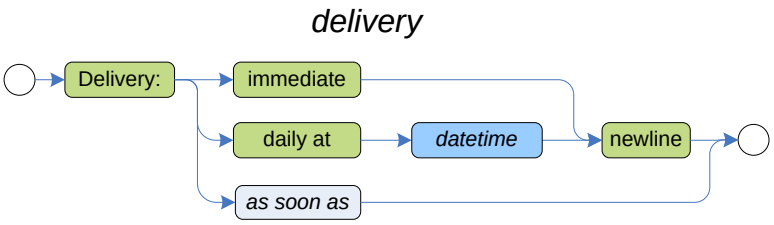

Figure 4.18: Delivery options

Figure 4.18 shows how the delivery option can be specified. In case of "Delivery:

immediate", the notification will be sent as soon as the conditions in the context are true. The second option is, that the notifications are delivered to a receiver every day at a specific time, eg. "Delivery: daily at #13:00#". It is up to the *Channel Adapter* component to aggregate the notifications and deliver them in one batch. Future version will support more complex delivery patterns (eg. every second Monday).

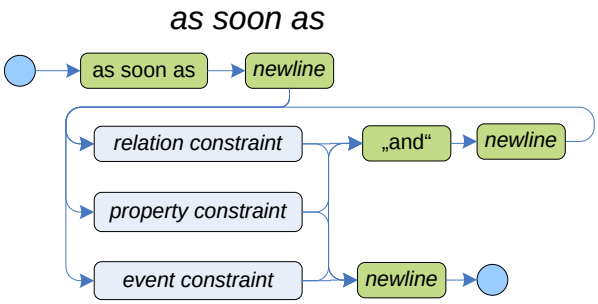

Figure 4.19: Conditioned delivery

The third option is to delay the delivery of a notification until some other conditions are met. The possibilities to define the condition resemble exactly the possibilities to describe the context as shown in figure 4.19. Variables that have been bound in the context specification keep their value as demonstrated in the following example:

```
1 Receiver: any User ?user
2 Context:
3 a Requirement ?requirement changes
4 Deliver: as soon as
5 the ?user "uses" the ?requirement
```

This specifies that users should get informed about requirement changes as soon as they start using the requirement, even if the event had happen in the past. Administrators can specify a timeout on how long an undelivered notification will be held in memory before it gets discarded.

4.5.6 Channel specification

The channel defines how a notification should be delivered to a receiver, eg. by Email or by Eclipse. For the *Rule Engine* component the channel does not matter

because it is not responsible for the actual delivery. The information is only relevant for the routing rules defined by the administrator and maybe the channel adapters.

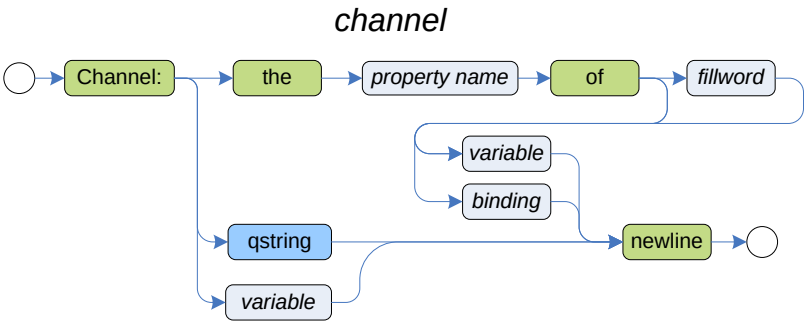

Figure 4.20: Delivery channel

As shown in figure 4.20 the channel can be specified by either 1) a quoted string, 2) the value of a variable, or 3) the property value of an artifact. Variables that were bound in one of the previous sections keep their value as demonstrated below:

```
1 Receiver: any User ?user
2 ...
3 Channel: the activeChannel of the ?user
```

The specification shown above sets the `channel` property of the notification to the `activeChannel` property of its receiver.

4.5.7 Message content

The *batch subject*, *subject*, and *body* specify the content of the notification. The *batch subject* is only used when the delivery option has been set to "daily at ...", and it is used by *Channel Adapter* components as the subject of the batch notification. The *subject* is compulsory and usually defines the message that should be displayed to the receivers upon delivery. The *body* is optional again and normally displayed only if the receiver requests more information eg. by clicking on the notification.

The content of the *batch subject*, the *subject*, and the *body* can be any qualified string (see figure 4.21) although only the *body* content is allowed to be multi-line. Within the string markers can be used fore referencing variables that have been bound in

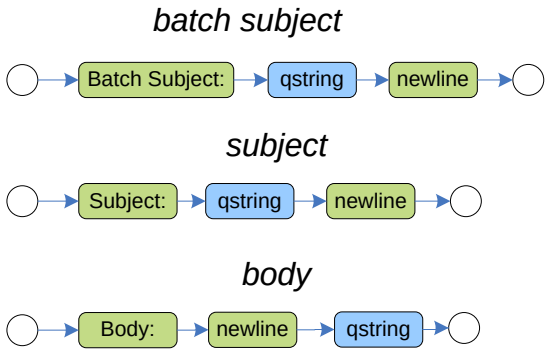

Figure 4.21: Notification content

previous sections. On delivery, the markers are replaced by the actual values as shown in the following example:

```
1 Receiver: any User ?user
2 ...
3 Subject: "Hi {?user.name}!"
```

The subject for a notification according to this specification could be: ”Hi Stefan!”.

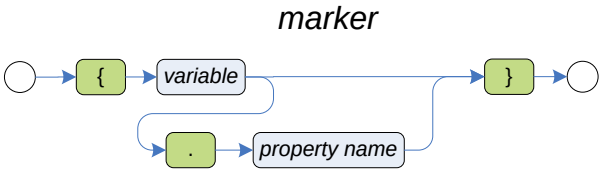

Figure 4.22: Notification content

Figure 4.22 shows the syntax for markers. If the variable is unbound or the artifact that is bound to the variable does not have a property with the specified name, the *state* property of the notification object will be set to ”error” and the markers left unchanged.

5 Case Study

This chapter shows on a simple scenario how the notification discovery and description framework and process, and NOTICON can be used in practice. It is focused on requirement changes which were identified as a key issue for many software development projects [8]. The scenario has been prepared in cooperation with experts from Siemens PSE.

The goals of the case study were:

1. To demonstrate how the notification discovery and description framework and process is used in practice.
2. To highlight some of the capabilities of NOTICON.
3. To give an expression of the user experience.
4. To better estimate the expected benefit of NOTICON compared to conventional notification technologies.

Our expectations were that we can easily apply our process in practice and discover communication issues that we did not think of beforehand. We also expected that we can show the usefulness of NOTICON even in a relatively simple project setting.

In section 5.1 we will give a short introduction of the project setting, followed by four sections were we will apply our process to 1) describe the communication risks, 2) describe the project organization, 3) describe the key communications, and 4) define the notifications (see section 3.5 on page 58). In section 5.6 we will show some screenshots that should give an expression on the user experience of NOTICON.

5.1 Introduction

Transporticon is a large software development company with its headquarter in Vienna that develops software applications in the line of transport and logistics. It has several hundred employees and depending on a project's requirements, the project manager can assemble a set of teams from the company's team pool. Each team is normally specialized on some tasks (eg. database design) and, following SCRUM best practices, consists of no more then 10 people [69].

Birgit is software project manger of *Transporticon* and currently responsible for developing a large software system for a container terminal in the Netherlands. The demands on the software are very challenging and the business rules very complex. Several hundreds of requirements have already be elicitated and Birgit expects the number to grow over thousand.

Birgit knows about potential communication risks in such a project setting and thinks that some of the risks could be mitigated with notification tools. Her expectations on a notification solution are an increased productivity of the developers and to be better able to communicate changes to impacted persons. Birgit decides to apply the notification discovery and description framework and process to approach the communication issues in a systematic way.

5.2 Project Communication Risks

The first step of the process is to analyze the project communication risks. Birgit quickly walks through the list of risk drivers (see 3.5.1 on page 62) and brainstorms on concrete risks that may be caused by the drivers and on their impact on the win conditions of the stakeholders. She also applies a simple ABC rating on how relevant each of the risk drivers is for the project and if a notification solution can help in mitigating them:

- *dependencies - Dependencies among elements that are managed by different teams*: Birgit knows that a requirement is usually implemented by several

components and by different teams. This creates dependencies between components that the developers might not be aware of. Birgit thinks that this might cause problem when the components get integrated. She expects this to endangers the schedule thus a risk mitigation is desired. The risk seems to be high enough for Birgit to deserve an A ranking, but she is not sure yet how a notification solution can help; therefore, she ranks its applicability with B.

- *changes - Frequent changes of elements that impact several teams at once*: Birgit immediately thinks of requirements that are created by the requirements management team and used by the configuration management team, the testing team, and the development teams. She already knows that the requirements are very unstable and change frequently (about 30 times per day). If these changes are not communicated in time to all involved parties, much time may be spent by the teams to rework something because of inconsistencies. The project has a very tight schedule which can be in danger if such situations occur too often; therefore, Birgit rates this risk driver with A. She is also sure that notification solutions can help in the risk mitigation and rates its applicability with A.

- *geography - Geographical separation of teams*: Some of the teams have their offices in Vienna and the others in Amsterdam. Birgit reckons that this may cause some inefficiencies in the communication between the teams. She senses the risk that important information is only communicated within one location and not between them. This can cause frustrated developers that feel uninformed about what is going on in the project. Birgit rates the risk driver and the applicability of a notification solution with B.

- For the other risk drivers Birgit could not think of any concrete risks so she does not investigate them any further.

Table 5.1 summarizes the relevancy and applicability of the risk drivers.

driver	relevancy	applicability
dependencies	A	B
changes	A	A
geography	B	B
other	C	C

Table 5.1: Rating of the risk drivers.

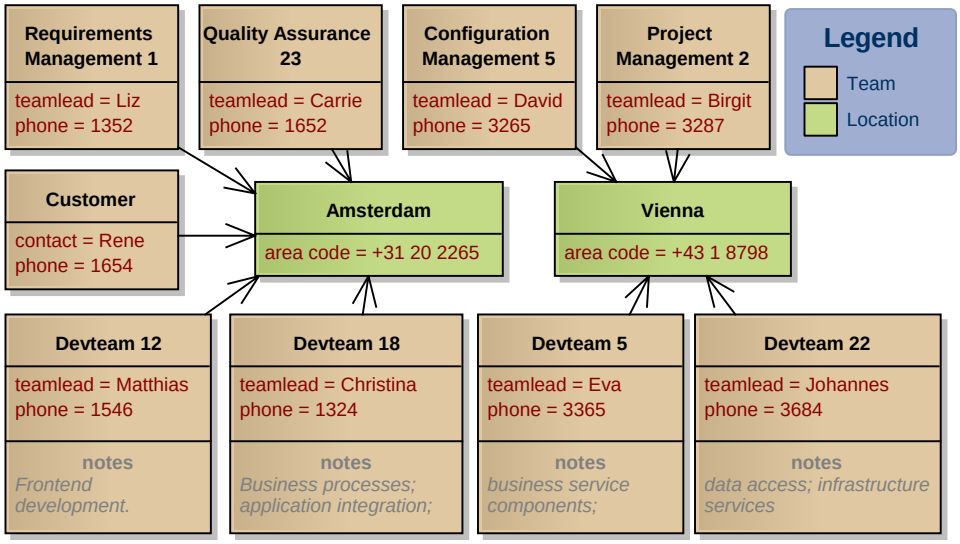

Figure 5.1: The people viewpoints shows the project organization.

5.3 Project Organization

After Birgit decided that some of the communication risks should and could be mitigated with a notification solution, she describes the project organization. Most of the information she visualizes has already been described in detail in the configuration management plan, so there is only little extra effort.

In the people viewpoint shown in figure 5.1 Birgit merely lists the teams she assembled for the project and includes the team leader's name and his or her phone number. She also highlights where the teams are geographically located because of the expected impact on the communication effectiveness. For the development

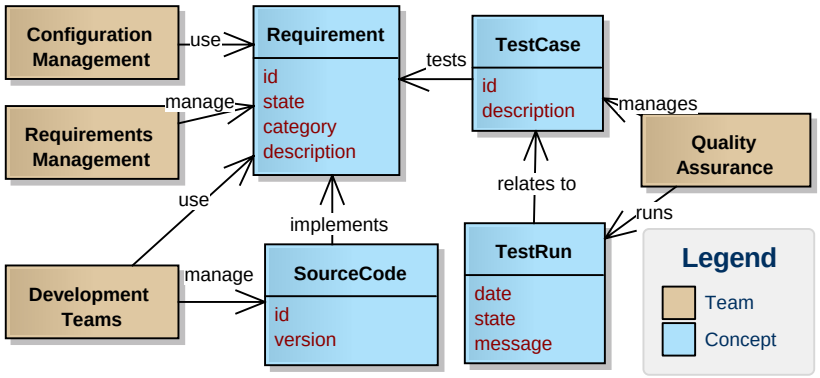

Figure 5.2: Information viewpoint focused on requirements.

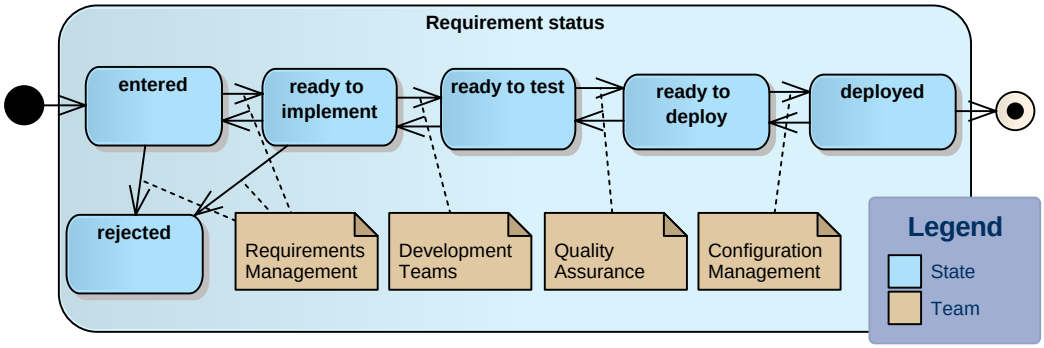

Figure 5.3: State machine for requirements.

teams Birgit also comments shortly on their main tasks. She decides not to include additional aspects like the language, culture, attitudes, etc. in the diagram because there is no indication that these issues could cause problems, and the related risk drivers are all C ranked.

In the information viewpoint, shown in figure 5.2, Birgit focused on concepts that are closely related to requirements. The trace between source code and requirements has to be maintained by the developers.The quality assurance creates at least one test case for each requirement and each test case can be run several times. Birgit also included all properties that she later may want to use in the notification specifications.

From previous projects Birgit knows that state changes of requirements are not communicated very well between teams; it lead to delays in the past because people

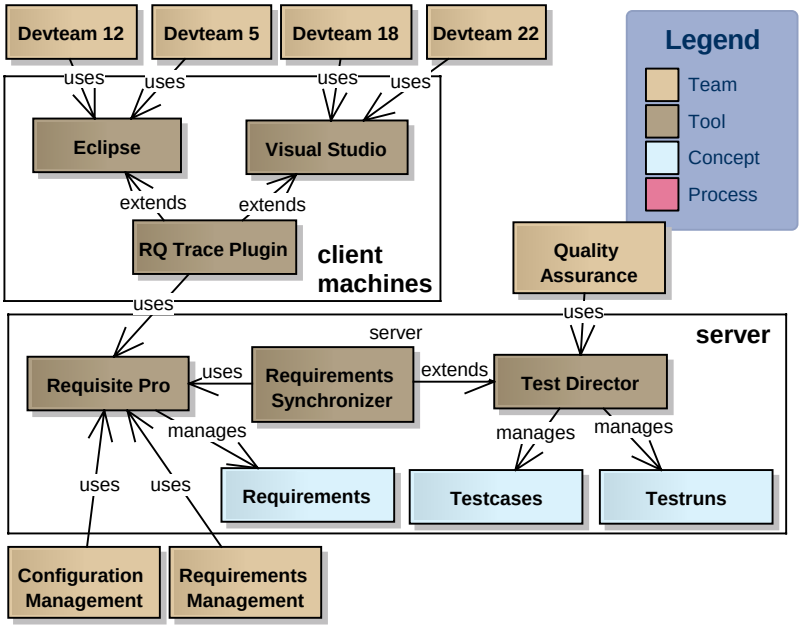

Figure 5.4: Technology viewpoint.

were waiting for something that has already happened. Therefore, she visualizes the state machine of the requirements and annotates the connections with the teams that are responsible to approve the next change (see figure 5.3). The diagram is also very useful to discover notifications that are related to requirement changes.

The technology viewpoint (figure 5.4) shows the tools that are used and the teams that primarily interact with them. Birgit decided to differentiate between the tools that are installed on the client machines and those that are installed on a server and accessed via a web browser. The *RQ Trace Plugin* extends Eclipse and Visual Studio and allows developers to efficiently link requirements to source code. The *Requirements Synchronizer* is an add-in for the Mercury TestDirector that synchronizes between the two systems on a predefined schedule [4]. The "manages" relation between tools and concepts denotes the master of the data; in case of synchronization issues the information from the master has precedence.

Figure 5.5 shows the "Implement Requirement in Eclipse" process that Birgit defined within the process viewpoint. It has a requirement that is in the state "ready to implement" as input which is changed to the state "ready to test" after the process

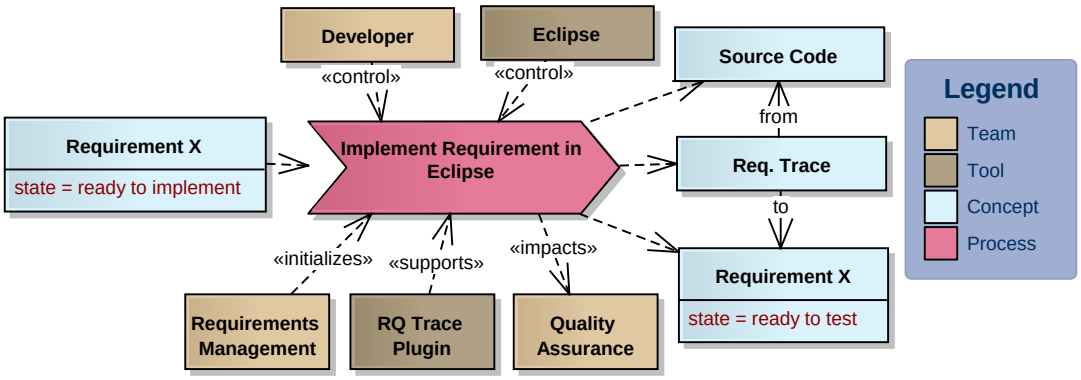

Figure 5.5: Process viewpoint showing the requirements implementation process.

completes. The process also outputs a number of source code artifacts that are annotated with tracing information. It is performed by developers within Eclipse and supported by the RQ Trace Plugin that creates the traces between source code and requirements. The requirements management team has to initiate the process. The outcome of the process impacts the quality assurance team because it needs to test the functionality.

5.4 Key Communications

In the next project step Birgit analyzes the viewpoints she developed in the last step and derives a list of key communications that are necessary for the project and might be optimized with a notification solution.

Birgit received some feedback from developers which complained about receiving too many emails about failed test cases that are not related to them. A person of the quality assurance team who runs a test case, manually creates an email if the test run fails and sends it to all developers. Typically many test runs are executed every day in parallel by several persons and many of them fail. This results in a very high number of emails. Listing 5.1 shows how Birgit described this communication act:

```
1 performer: members of the Quality Assurance team
```

```
receiver: Development teams
intention: directive to correct the code
message: The test case X testing the requirement Y failed
time-for-completion: depending on the release plan
medium: email
trigger: a test run testing the test case X fails
frequency: about 50 times per day
```

Listing 5.1: Quality Assurance communicating failed test cases

Another communication act that Birgit described is shown in listing 5.2:

```
performer: Requirements Management
receiver: Customer Representatives
intention: information about requirements that could be delivered
message: the following requirements are ready to deliver: Req. X, Req. Y, ...
time-for-completion: undefined
medium: phone or email
trigger: the state of a requirement changes to "ready to deploy"
frequency: 1 time per day
```

Listing 5.2: Requirements Management communicating requirements that are ready to deploy

Although the communication act does occur only once a day, Birgit decided to describe it in detail because if it is overseen it can lead to conflicts with the client.

```
performer: Developer
receiver: Developer
intention: information that they are working on the same requirement
message: developer XXX is also working on requirement 15
time-for-completion: undefined
medium: development environment
trigger: two developers concurrently work on the same requirement
frequency: about 5 times per day per developer (but does not happen at the moment)
```

Listing 5.3: Developers making them self aware of each other

The communication act described in listing 5.3 is one that did not happen yet, but is considered as relevant by Birgit. Two developers that are concurrently working on the same requirement should be made aware of each other. Birgit expects that an increased awareness reduces incompatibilities of the source code because developers can quickly get in contact with each other in case of questions.

5.5 Notification Definitions

After Birgit defined the key communications, she formalizes the notifications that should be sent. The example we gave in section 3.5.4 on page 74 already showed how a notification for changing requirements can look like; thus, we describe some slightly more complex notification scenarios that demonstrate the capabilities of NOTICON:

5.5.1 Failed test-case

Description: If a test case fails, all developers that are subscribed to this notification and recently worked on source files that implement the test case should be informed in the tool they are currently working with.

Scenario: The test case with the id 23 tests the requirement with the id 12. It is implemented by two source files; one created by Chloe and the other by Stephan. If the test fails, it triggers an immediate notification for Chloe and Stephan.

Upsides: Developers can quickly react on failed test cases if they are currently working with Eclipse. It is expected that this reduces the average time it takes that errors get fixed.

Downsides: Too many false positives would slow down the primary activity.

Notification Specification:

```
Title: "Notify developers of failed test cases."
Receiver: any subscribed User ?user
Context:
the ?user "has" the Role "developer" and
the ?user "uses" a Requirement ?requirement and
a TestCase ?testcase "tests" the ?requirement and
a TestRun ?testrun "relates to" the ?testcase and
the status of the ?testrun changes from "undefined" to "failed"
Deliver: immediate
Channel: the activeChannel of the ?user
Subject: "Test case {?testcase.id} failed."
Body:
```

```
13 "{?testrun.date}: The test of requirement {?requirement.id} failed
14 in the test run {?testrun.id}"
```

Remarks: In line 2 we specify that users have to subscribe to get notifications created by this specifications. Line 4 states that a user that receives the notification has to be a developer. In line 5 we bind the `?requirement` variable to all requirements the user is currently using. The information which requirements a user is using is gathered by sensors in Eclipse and Visual Studio. In line 6 we check if there is a testcase that tests the requirement and binds it to the variable `?testcase`. Then we specify that there has to exist a test run that relates to the test case. In line 8 we finally specify that an event occurs that changes the status property of the test run from "undefined" to "failed". Only if all the conditions are fulfilled, the notification will be generated. It should to be delivered immediately (line 9) via the channel that is stored in the `activeChannel` property of the user (line 10).

Implementation Issues: 1) The relationships between source code and requirements must be traced. 2) The relationship between requirements and test cases must be traced. 3) The relationship between test cases and test runs must be traced. 4) It must be possible to sense the status of test runs. 5) It must be possible to sense the relationship between a user and a requirement.

5.5.2 Daily summary of requirements that are ready to deploy

Description: The customer and the configuration management team should receive a summary about the test-cases that have the status "ready to deploy" every day.

Scenario: Lukas is the customer representative who decides which requirements should be delivered in which release. Every day at 9:00 he receives an email of all requirements that changed their state to "ready to deploy"; thus, they can be scheduled by Lukas for one of the next releases.

Upsides: 1) It is not required any more, that the customer gets access to the requirements management tool. 2) The customer gets informed automatically making the daily summary emails that would have been created otherwise manually by the

requirements management team obsolete.

Downsides: No additional information can be included in the email.

Notification Specification:

```
1  Title: "Notify customer of requirements that are ready to deploy."
2  Receiver: any User ?user
3  Context:
4  the ?user "has" the Role "customer" and
5  the status of a Requirement ?requirement changes to "ready to deploy"
6  Deliver: daily at 09:00
7  Channel: "Email"
8  Batch subject: "Daily status report."
9  Subject: "Requirement {?requirement.id} is ready to deploy"
10 Body:
11 "{?requirement.description}"
```

Remarks: In line 4 we specify that only users who have the role "customer" should receive the notification. In line 5 we wait for an event on any requirement that changes its status to "ready to deploy". Line 6 shows that the notifications should be aggregated and sent in one email (line 7) every day at 9:00. If there are no requirements that have changed to "ready to deploy", no notification will be sent. In line 8 we specify the subject that should be used for the batch of notifications. It is up to the channel adapters (see 4.1 on page 77) to interpret this information correctly.

Implementation issues: 1) Customer representatives have to be registered as users. 2) It must be possible to sense the status of requirements.

5.5.3 Working on the same requirement

Description: If two persons are working on the same requirement, they should get informed about each other.

Scenario: Agnes from the "Devteam 18" implements a functionality for the requirement 887. At the same time Peter from the "Devteam 22" is also working on requirement 887. Immediately they get a notification in their development environ-

ment that they are concurrently working on the same requirement.

Upsides: It is expected, that the increased awareness between the developers enhances their productivity by reducing the possibility of versioning conflicts and reworks.

Downsides: If many developers are concurrently working on the same requirement, too many notifications may be sent which slows down their work.

Notification Specification:

```
1 Title: "People are working on the same requirement."
2 Receiver: any subscribed User ?user
3 Context:
4 the ?user "uses" a Requirement ?requirement and
5 a User ?other "uses" the ?requirement
6 Channel: the activeChannel of the ?user
7 Subject: "{?other.firstname} {?other.lastname} ({?other.phone}) is
8           also working on the requirement {?requirement.id}"
```

Remarks: In line 2 we specify that the users have to subscribe to receive these notifications. In line 4 the variable `?requirement` is bound to all requirements a user is currently using and line 5 defines that all users which work on the same requirement should be bound to the variable `?other`. Line 6 states that the information should be sent to the channel that is specified in the `activeChannel` property of the user artifact. This allows people to get notified about each other, even if they are working in different development environments.

Implementation Issues: 1) The requirement a user is currently working on needs to be sensed.

5.6 User Interaction

Once installed, NOTICON integrates seamlessly into the user's working environment and does not require manual interaction. Only when a user wants to change some preferences (eg. subscribe or unsubscribe from or to notification specifications), or add a new notification specifications, the administration component shown in figure

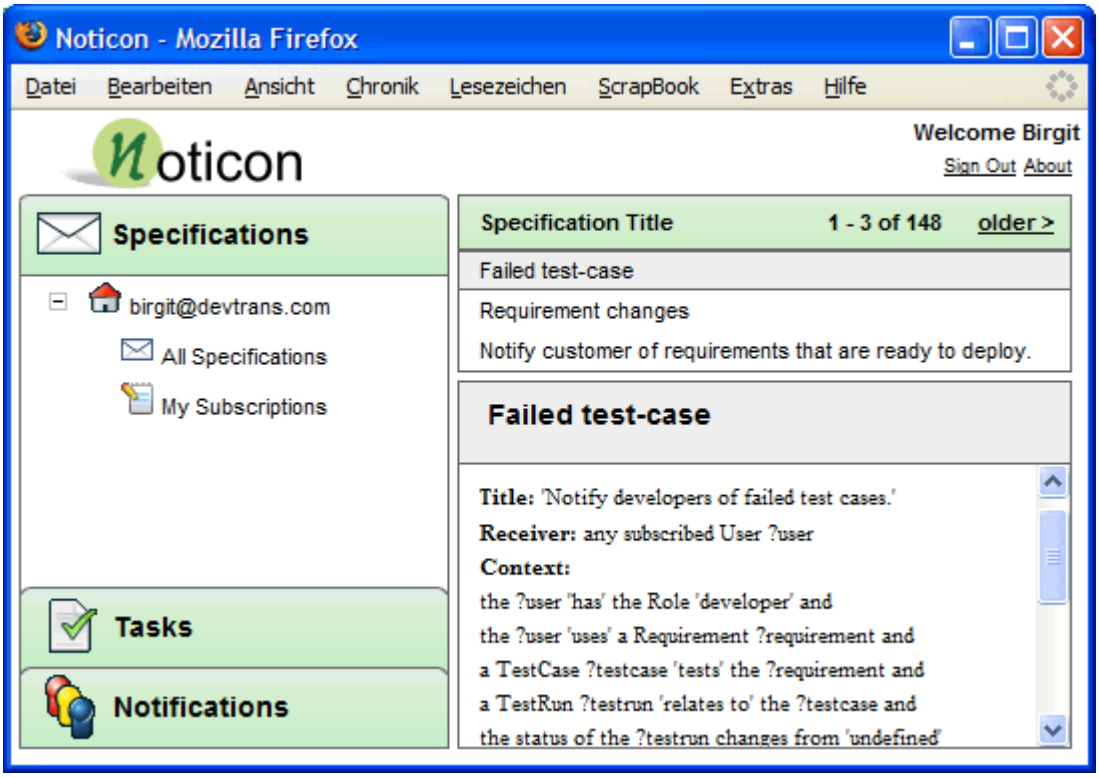

Figure 5.6: Aministration application for NOTICON.

5.6 needs to be used.

In comparison to other notification systems NOTICON displays notifications in the tools the user is currently using and is sensitive to his or her current work context.

In figure 5.7 we demonstrate how the notifications defined further above are displayed to developers. The upper part of the figure shows the integration into Eclipse and the lower part into Visual Studio. We extended both programs with a notification panel which lists all notifications that were generated for the user and with the channel property set to "Eclipse" or "VisualStudio". On double-click of a list entry, the details of the notification message are displayed.

In the upper part of Visual Studio and Eclipse the source code is displayed as usual, but the classes have been annotated with special comments that link them to requirements. Eg. the Java class "mytest" is linked to the requirement 14 and to the requirement 144. XXX developed a plug-in for Eclipse and Visual Studio, that

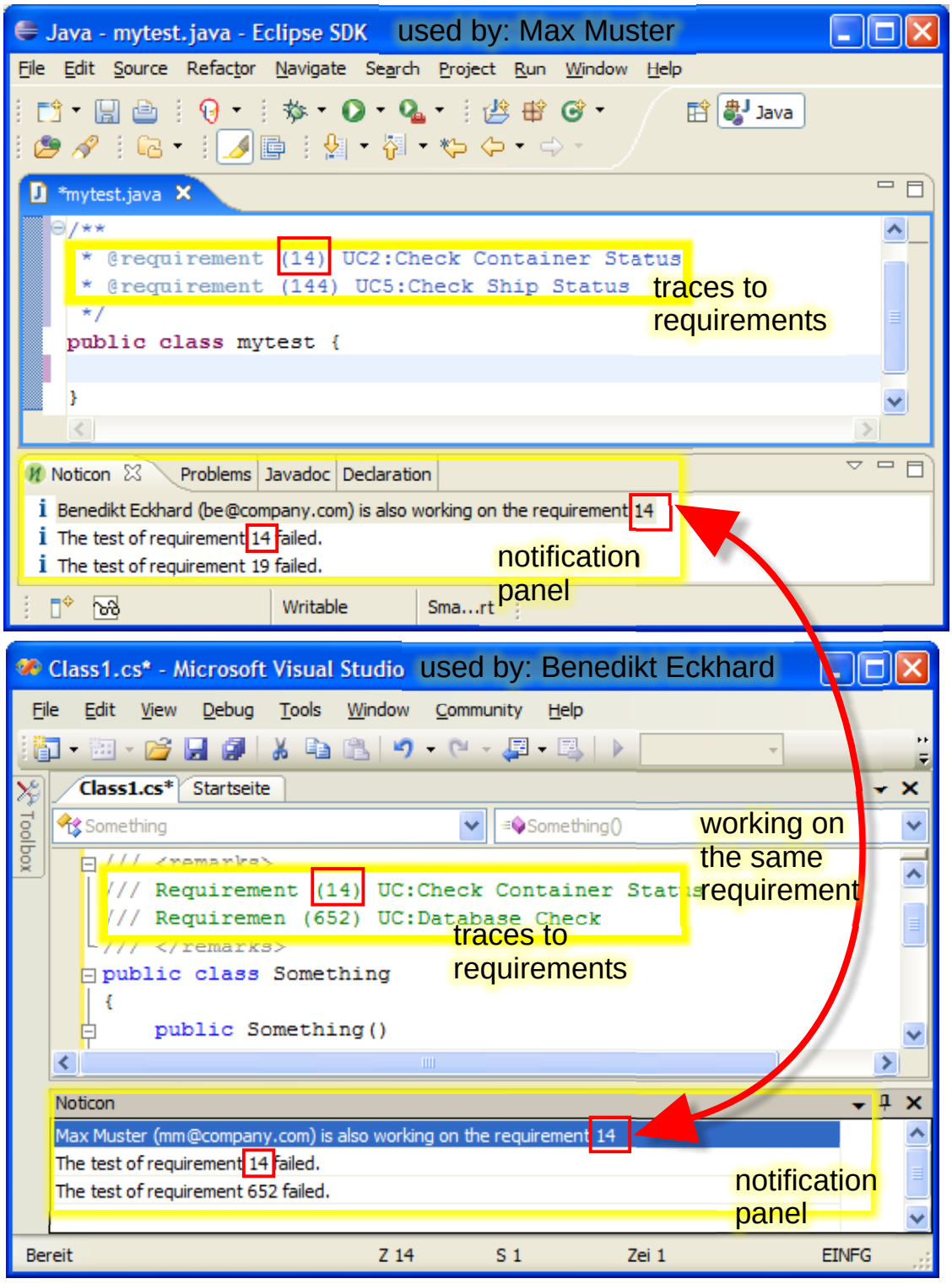

Figure 5.7: NOTICON embedded in Eclipse and Visual Studio.

can interpret these comments and create bidirectional links between source code and requirements that are managed in Requisite Pro. It can also display a filtered list of the requirements within the development environments; to create traces, users just need to click on an item in the list and the comment is created automatically for them. NOTICON uses the information created by the plug-in and makes it available in the context model.

In our example, both the Java class "mytest" and the C# class "Something" are related to requirement 14. Max edits the Java file and Benedikt the C# file at the same time, which triggers the "Working on the same requirement" notification as specified above. Immediately, Max gets the information that Benedikt is also working on the requirement into his notification panel and vice versa.

The two other notifications that are displayed in the notification panel of Max and Benedikt are related to failed test cases and triggered by the "Notify developers of failed test cases" specification. Both get the information about the failed test case of requirement 14 but only Max receives the information about requirement 19 (which he has worked on recently) and only Benedikt gets the information about the requirement 652.

Users have the possibility to right click on a notification to unsubscribe from this notification type. In the future we plan to provide users with the functionality to tailor the notification specification from within the tool that the notification was sent to.

Figure 5.8 shows an example of the email that is sent every day at 9:00 to all customer representatives according to the "Daily summary of requirements that are ready to deploy" notification specification defined on page 114.

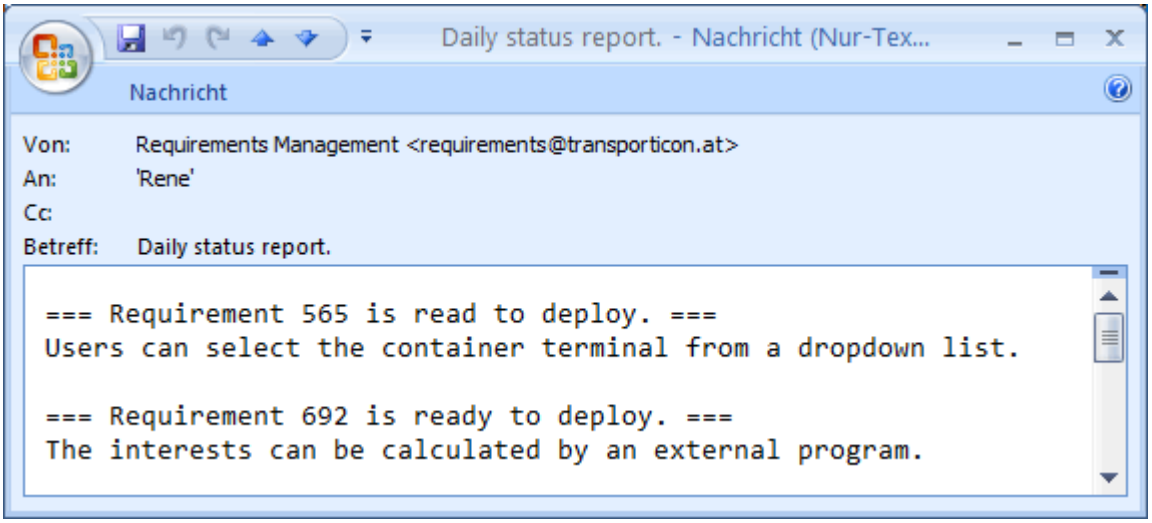

Figure 5.8: Daily status report for customer representatives

6 Discussion

This chapter discusses the results of this work from several perspectives. Section 6.1 describes a total cost of ownership analysis where the impact of our solution is compared to conventional notification systems on the example of the case study that was described above. In section 6.2 we will discuss the process and section 6.3 critically looks at the pros and cons of the prototype. The notification specification language will be discussed in section 6.4 and finally the expected benefits and costs of NOTICON are discussed in section 6.5

6.1 Total Cost of Ownership

The cost for introducing a notification system like NOTICON is certainly higher then that of a conventional solution; thus, estimating the benefits is critical. We developed a model based on the "Total Cost of Ownership" (TCO) [45] method that we think is suitable for comparing different notification approaches. We applied it to a setting we deduced from the case study described above. The setting and assumed numbers are discussed with experts from Siemens PSE.

6.1.1 Setting

Our setting is a software development project with 70 developers ($n(emp)$) and an average of 30 requirements change per day ($n(change)$), for 50 days ($n(dur)$). The probability that a change concerns a developer is 10% ($p(change)$) and the propability that a change would require an immediate reaction is also 10% ($p(react)$).

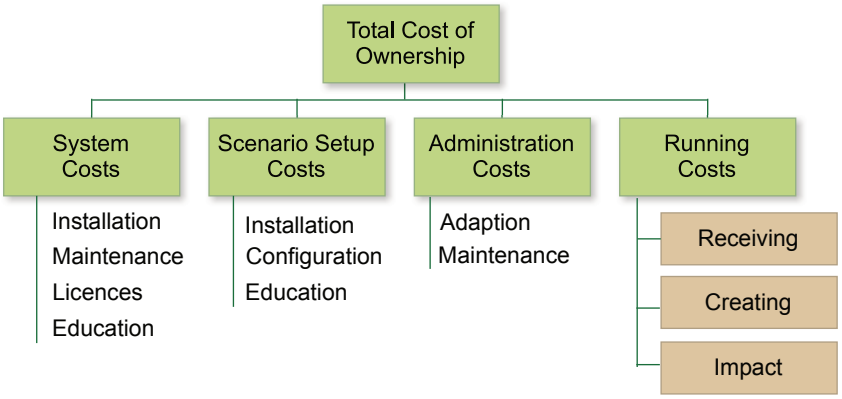

Figure 6.1: Total Cost of Ownership Model

We estimated the cost for an employee with €50,– per hour ($c(emp)$) and that of an external consultant with €1500,– per day ($c(cons)$).

Two notification alternatives are compared:

1. The use of the email notification mechanism that is built into the requirements management system. On every change it creates an email that gets delivered to all developers.

2. The use of NOTICON, where developers get notified within their development environment about changes, but only if the change concerns them. A change is assumed to concern a developer if he or she has recently been working on code that relates to the requirement. We assumed that the relations between requirements and code have to be maintained regardless of the notification approach; therefore, we did not take that effort into account.

6.1.2 Model

As illustrated in figure 6.1 we calculate the TCO by estimating the costs along four main categories: System Costs ($c(sys)$), Scenario Setup Costs($c(scen)$), Administration Costs($c(admin)$), and Running Costs ($c(run)$). Formula 6.1 shows the calculation of the TCO.

$$TCO = c(sys) + \sum_{\forall s \in S} (c(scen, s) + c(admin, s) + c(run, s)) \tag{6.1}$$

System Costs occur regardless of the concrete scenario and only once. *Scenario Setup Costs* occur once for each scenario (s in S). *Administration Costs* are costs for a scenario that can occur for each user and *Running Costs* estimate the costs for the notifications.

The next sections will describe the different categories in detail and estimate concrete values for the two alternatives.

System Costs

The **System Costs** are costs that occur once for the project regardless of a concrete notification scenario and are divided into four categories:

- *Installation Costs* include the cost for the system itself, and the additional costs that are necessary to install the system.

  NOTICON is an open source product; thus, no installation costs occur. We assume that the installation is done by a consultant who is supervised by a system administrator of the company within one day. This leads to estimated installation costs of €1.900,– ($c(emp) * 8 + c(cons) = 50 * 8 + 1500$).

  In case of the email notification, a plug-in needs to be installed into the requirements management system. This can be done by the system administrators in a four hours; therefore, we assume costs of of only €200,– ($c(emp) * 4 = 50 * 4$).

- *Maintenance Costs* include costs for server restarts or the installation of software when a new computer is added to the network.

  In our setting we assume that NOTICON requires 8 hours maintenance time for the whole project duration; thus, the maintenance costs are estimated with

€400,– ($c(emp) * 8$).

For the email notification we do not assume any maintenance costs as they cannot be differentiated from the maintenance costs of the requirements management system itself.

- *License Costs* include the costs for software licenses that have to be obtained to run the notification solution.

  A simple email notification of most of requirements management tools just contains the information that a requirement has changed, but does not include any details (eg. which fields have been changed, the old and the new value, etc.). To get the details users typically have to log into system. Depending on the license terms of the application either all users have to have a license, or floating licenses can be obtained that permit a defined number of concurrently logged in users. Changes are communicated to several developers at the same time, and we expect that many of them check for details immediately after receipt. We therefore expect an increased number of the required licenses because of notification systems.

  Notifications created with NOTICON can include many detail of the requirements change, and also the amount of notification that a change triggers is reduced; therefore, we assume that only 5 additional floating licenses need to be purchased for the requirements management system. We estimated the price of one license with €2.000,– which leads to license costs €10.000,– ($5 * 2000$).

  For the email alternative we assume that 20 additional floating licenses need to be purchased because all 70 developers are notified of a change at the same time; thus, the license costs are €40.000,– ($20 * 2000$).

- *Education Costs* include the costs and time that it takes to learn how to work with a system. For notification systems the learning costs typically occur only for the administrators of the system and not for each user.

  NOTICON has a very rich set of functionality and administrators will likely take a three day course to understand the system and all configuration options.

	email	NOTICON
Installation	200,–	1.900,–
Maintenance		400,–
License	40.000,–	10.000,–
Learning	200,–	4.200,–
Total	**40.800,–**	**16.500,–**

Table 6.1: System Costs in €

We estimated the costs for such a course with €3.000,–. Additional €1.200,– ($c(emp) * 8 * 3$) are calculated for the working time of the administrator. This results in education costs of €4.200,– (1200 + 3000).

The functionality of email notification plugins of requirements management tools is often very restricted and easy to learn. We expect that it can usually be learned by an administrator in half a day; thus, we assume costs of only €200,– ($c(emp) * 4$).

Table 6.1 summarizes the expected system costs. The email per change solution has much higher system costs, but only if additional licenses have to be purchased. Otherwise the initial investment of NOTICON is about seven times higher than that of a traditional approach.

Scenario Setup Costs

Scenario Setup Costs describe the costs that are necessary to setup a concrete notification scenario (eg. sending an email when a requirement changes). We differentiated them from the *System Costs* because usually several scenarios can be realized with one installation. The costs are estimated along three categories:

- *Installation Costs* are the costs for additional software that needs to be installed to support a concrete scenario.

  NOTICON requires a sensor to be installed in all development environments

that senses the relation between requirements and users. We estimated that this takes 10 minutes for each developer which sums up to about €580,– in our setting ($n(emp) * 10 * \frac{c(emp)}{60}$).

For the email alternative no additional installation costs to setup the scenario are assumed.

- *Configuration Costs* include the costs that are necessary to configure systems to support the scenario.

  For NOTICON a context model needs to be created and the routing rules configured correctly. We assume that this is done by an external consultant in one day and estimate €1.500,– ($c(cons)$).

  For the email notification a mailing list needs to be created, and all developers have to subscribe to this list. We assume that this takes about 5 minutes per developer resulting in estimated costs of €290,– ($n(emp) * 5 * \frac{c(emp)}{60}$).

- *Education Costs* are the costs that it takes to learn how the concrete notification scenario works. This is only relevant for users that want to adapt the notifications according to their preferences.

  NOTICON we assume that about 10% of the developers want to change preferences. Understanding the notification specification language and the context model for this scenario takes at an outside estimate 2 hours; thus, we estimate total costs of €700,– ($0.1 * n(emp) * 2 * c(emp)$)

  In case of the email scenario users cannot set preferences individually; therefore, no education costs are calculated.

Table 6.2 summarizes the scenario setup costs and shows that another approximately €3.000 are required to implement the scenario with NOTICON. This is significantly more than the scenario setup costs for the email alternative.

	email	NOTICON
Installation		580,–
Configuration	290,–	1.500,–
Education		700,–
Total	**290,–**	**2.780,–**

Table 6.2: Scenario Setup Costs in €

Administration Costs

Administration Costs are costs that occur during the project lifetime because either the project changes (eg. a new developer enters the project) or users change their preferences. They are divided into two groups:

- *Adaption Costs* occur when users change their notification preferences, or ask an administrator to do so.

  10% of the developers are expected to change their preferences in NOTICON. A change is assumed to take about one hour; therefor, the adaption are €350,– $(0.1 * n(emp) * c(emp))$.

  Email notifications usually do not support an individual adaption and no additional costs are assumed.

- *Maintenance* costs include the costs that emerge when users would be required to constantly interact with the notification system (eg. if relations between artifacts needed to be established manually).

  In our setting neither the email notification nor the notification with NOTICON requires special attention by the user; thus, no maintenance costs arise.

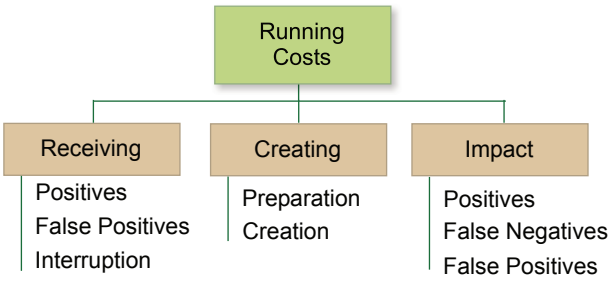

Figure 6.2: Running Costs

Running Costs

The **Running Costs** try to estimate the notification's impact on the receivers. We divided them into three groups, each with several cost factors as shown in figure 6.2:

- *Receiving Costs* occur for each user on each notification he or she receives.
  - The *Interruption* estimates the effort for switching the the attention from the primary task to the notification and back to the primary task.

    When using NOTICON the interruption effort is estimated to be 30 seconds per received notification. As described in section 6.1.1, we assume 30 changes per day and a probability of 10% that a change concerns a developer. It is expected that every developer receives only one false positive per day when using NOTICON; thus 280 interruptions occur per day $(n(emp) * n(change) * p(change) + n(emp))$. This leads to daily interruption costs of €117,– $(280 * n(emp) * 0.5 * \frac{c(emp)}{60})$.

    In case of the email notification the interruption effort is estimated to be 1 minute. It is higher than that of the NOTICON alternative because NOTICON can display the notifications in the application the receiver is currenly using. Changes of requirements are broadcasted to all developers; thus, the interruptions occur 2100 times per day $(n(emp)*n(change))$ leading to daily interruption costs of €1.750,– $(2100 * n(emp) * \frac{c(emp)}{60})$.
  - *Positives* measure the effort to read and make sense of a notification that

is relevant for a user.

NOTICON messages can contain detailed information about the change, and usually no login to the requirements management system is required; thus, we assume an effort of 30 seconds per notification. This leads to daily costs for positives of €88,– $(n(emp) * n(change) * p(change) * 0.5 * \frac{c(emp)}{60})$.

In case of the email notification the user has to login into the requirements system to get all relevant information; thus, we assume 3 minutes reading costs for positives resulting in daily costs of €525,– $n(emp) * n(change) * p(change) * 3 * \frac{c(emp)}{60}$.

– *False Positives* measure the reading effort for notifications that do not concern a developer. Like with SPAM emails, it usually takes some time to recognize if a information is of value.

With NOTICON we calculated an effort of 30 seconds for false positives. Assuming 1 false positive per day per developer, the daily costs would be €29,– $(n(emp) * 0.5 * \frac{c(emp)}{60})$.

In case of the email scenario the costs for false positives are assumed to be 1 minute. Occasionally a login into the requirements management system might be required to check if it is a false positive or not. 1890 false positives are sent every day $(n(emp) * n(change) - n(emp) * n(change) * p(change))$ resulting in daily costs for false positives of €1.575,– $(1890 * \frac{c(emp)}{60})$.

- *Creating Costs* are costs that occur when creating notifications. For estimating them, both the *preparation* and the *creation* costs must be considered. Eg. if notifications are sent via SMS, the preparation costs would include the effort to write the message, and the creation costs would include the costs of an SMS. For the notification scenario at hand the creating costs are zero for both alternatives.

- *Impact Costs* try to estimate the impact of notifications along three categories:

– *Positives* describe the impact of a positive notification. This can be either done by estimating the benefits of the notification (expressing it as negative costs) or by estimating the costs that occur although the notification was sent.

  We calculated the scenario with the second approach: It was estimated that about 10% of the requirement changes that concern a developer would require an immediate reaction of him or her. Costs arise for any minute that developers are not aware of the change (eg. because they would have to rework their work package later on).

  With NOTICON changes are immediately communicated to developers within their development environment. This guarantees that no notification is overseen; thus, no costs for positives are assumed.

  In case of the email alternative, we assume a timespan of 15 minutes between the time a requirement gets changed and the email is read by the developer that is affected by the change. Having 21 requirements each day that require an immediate reaction ($n(emp) * n(change) * p(change) * p(react)$), the total daily costs for positives would be of €175,– ($21 * 15 * \frac{c(emp)}{60}$).

– *False Negatives* describe the impact of notifications that should have been delivered but were not.

  We assume that these conditions occur because of system errors and can be discovered by the system administrator (eg. because the email server had a problem) who will communicate this immediately. The likelihood of these situations is expected to be the same for both notification alternatives; therefore, we did not include these costs in the calculation.

– *False Positives* describe the impact of false positives. This is important if notifications would immediately trigger actions - eg. the build in a build server.

  In our setting false positives do not have any negative effects besides the

	email	NOTICON
Receiving: Positives	525,–	88,–
Receiving: False Positives	1.575,–	29,–
Receiving: Interruption	1.750,–	117,–
Impact: Positives	175,–	0,–
Total	**4.025,–**	**233,–**

Table 6.3: Running costs per day in €.

	email	NOTICON
System Costs	40.800,–	16.500,–
Scenario Setup Costs	290,–	2.750,–
Administration Costs		350,–
Running Costs	201.250,–	11.650,–
Total	**242.690,–**	**31.250,–**

Table 6.4: Total costs for a period of 50 days in €.

additional interruption and reading costs that have already been taken into account.

Table 6.3 lists the running costs PER DAY that we estimated for our scenario. Most of the difference is caused by the much smaller number of notifications that a developer gets when NOTICON is used. The total of the running costs for a period of 50 days are €201.250,– for the email alternative ($50 * 4025$) and only €11.650,– for NOTICON .

6.1.3 Conclusion

The total costs for are period of 50 days are about €240.000,– for the email alternative and €31.000,– when using NOTICON. Expressing that in minutes means that every developer spends about 80 minutes reading notification emails every day,

compared to 10 minutes they would spend with NOTICON.

This clearly shows the benefit of using complex notification rules in projects where many notifications need to be sent every day.

But the initial investment for NOTICON, although it is an open source project, should not be underestimated: The costs for just installing the systems component and having the administrators trained are estimated to be about €9.000,–. This suggests that project manager should do a detailed calculation based on our TCO framework to estimate in advance whether NOTICON is worth the investment. In our scenario NOTICON breaks even with the email alternative after about about three days or 6300 delivered emails (if we do not consider the extra license costs that we expected).

6.2 Discussion of the Process

The notification discovery and description framework and process answered our research question on how notification needs in GSD projects can be discovered and described. It was designed to fulfill four goals:

1. It should support the discovery of communication problems and help in mitigating them by automating key communications in form of notifications.
2. It can be applied easily with the tools and methods that the project manager is already familiar with.
3. It can be tailored to the specific needs of a project.
4. It should be applicable with low budget and little time.

To validate these design goals we applied the process to a scenario that has been designed in cooperations with experts from Siemens PSE (see chapter 5).

One of the most important things we discovered was that the process must be applied iteratively. By following the process several ideas for one of the next process steps will pop up, and we suggest to follow those ideas first than to continue with the

current process step. Our experience with the case study showed that this leads to a quick output in all process steps which can be refactored and extended in subsequent iterations.

From our literature research on the characteristics of GSD projects (see chapter 2) we distilled a set of risk drivers that are described in detail in section 3.5 on page 58. Our experience with the case study highlighted their importance: they focused the other analysis steps on the aspects that were relevant for the setting. We argue that this is important, as otherwise too much unnecessary information would slow down the analysis.

Drawing the development project from the different viewpoints, as suggested in step 2, was actually more work than expected. To avoid getting lost in details it is important to focus on information that is relevant in relation to the risk drivers or should be generally known by all project members; our experience showed that these diagrams serve perfectly well for communicating important concepts to all project members. Especially in bigger projects those diagrams are often created for the configuration management plan. In that case they should be examined if they contain enough information concerning potential communication risks.

Another important observation regarding the viewpoints was that focus should be put on expressiveness rather than following standard notations like BPMN. The primary intention of these diagrams is to make discovering potential communication problems easier and not serve the base for automating business processes. Nevertheless we suggest to stick to notation that are well established within a company.

One of the most important viewpoints if NOTICON should be used as notification solution is the information viewpoint. If it describes all relations between concepts and their attributes, the mapping to a concrete context model is easy. Even Model Driven Architectures may be applied to generate the context model automatically. The information viewpoint also serves as a valuable reference for writing notification specifications.

In our opinion the usefulness of describing the communication acts in the structures way we propose depends on the project phase in which the notification solution gets

introduced. If the project has not been started yet, the communication acts can just be assumed and might not reflect the actual communication acts. This implies that the process outputs and notification specifications should be refined after the project has started.

6.3 Discussion of the Prototype

In this section we discuss issues regarding the prototypical implementation of NOTICON.

A key requirement we identified is the use of standard technologies (see section 3.4). We argued that this is important for administrators that have to maintain and configure the system. In practice it showed that standardization is not that relevant for administrators because they do not make changes to the system at that low level. Instead they voted for a more packaged approach, where the configuration can be done with wizards and installation assistants.

The use of standard technologies provided us with a significant boost in prototyping the system and, even more important, attracted other developers that wanted to join our efforts. Technologies like Mule, Drools, and ActiveMQ, have a very active community which we can leverage for continuous knowledge transfer.

Relying on an enterprise service bus as event and notification hub is in our opinion a significant differentiation to other approaches. The benefit we gained is that we are not limited to a single technology (eg. Java) and that connectivity from and to various external systems is much easier.

The event driven architecture with flexible routing logic carried out by the service bus promotes a very loose coupling of the systems components. In our opinion this encourages the participation of other developers as they can work relatively independent from each other and incorporate their extensions quickly. It also enables integration with other tools (eg. stream engines) that can process events before or after they are processed by the rule engine.

One issue that has to be proved in practice is, whether the rule engine approach can scale well enough for big projects. The number of artifacts and their relations that have to be held in the working memory of the rule engine depends on the complexity of the notification specifications, but can be huge. The current approach to associate artifacts with an expiration date might lead to undesirable side effects or do not provide enough flexibility. Further research and empirical evaluation is required to better understand these issues and find strategies to tackle them.

As highlighted by Booker et al. [12], privacy is a very important aspect for the success of a notification solution. In the current implementation we pushed this issue on the side, but are aware that it needs to be addressed in the near future. Currently there are no limits on what can be specified in a notification specification. It would be possible that the management uses the tool to spy on the activities of their employees. Strategies against this potential misuse have to be defined and could range from technical changes to organizational control of the notification specifications (eg. they could have to be approved by the shop committee).

Using a Java class based context model was a pragmatic decision rather then adopting suggestions found in literature. Strang and Linnhoff-Popien [81] showed that context models based on semantic technologies like RDF and OWL provide a more expressive and standardized way to manage concepts and their relations; thus, they should be preferred to class based models. But our evaluation of the existing semantic tools showed that the additional effort to use and adapt them to our requirements exceeds the benefits. This is mainly due to a fundamental difference in how the data is accessed: we require a model where changes in the data actively trigger notifications, but the semantic toolkits like Jena [1] currently only support efficient querying of data.

One of the key requirements we identified for a notification system was that it must not interfere with the primary work of the user. Thus, we embedded NOTICON into the tools that a user is currently using; notifications blend into the users environment and do not distract them. What we found as an advantage is that the logic of which notifications should be displayed resides on the server side and is not done by the

[1] http://jena.sourceforge.net/

presenters; this eases the integration into applications.

In the current version, NOTICON does not support queries of the context data. This functionality is very well supported by other context-aware systems, but was not the primary focus of this work. we plan to add query functionality in future versions because we think that the data that is maintained to provide the notifications can be used for other applications as well.

6.4 Discussion of the Notification Specification Language

The specification of notifications should be done by business users in the domain specific language that we specifically designed for this task. Our design goals were:

- It must be easy to understand and write by business users who have no prior knowledge in programming languages and/or rule engines.
- It must be expressive enough to support a wide variety of notification scenarios.
- It can be compiled relatively easily into the Drools specific rule format.

We evaluated the language with two potential users that had no prior knowledge of notification systems, nor of programming languages. Our findings were that reading and understanding notification specifications was possible without much introduction. Only the concept of variables had to be explained. What showed to be very beneficial were fillwords like "a", "an", "any", etc. that have no semantic meaning in the language but lead to better readability of the specifications.

While reading the language did not cause troubles, our test users were not able to specify notifications themselves, giving them only the context model and the language specification as references, within a short time. The initial only textual representation of the language grammar was way too abstract to be understood by non-programmers and also the graphical notation changed this only slightly (at least

programmers were able to understand it then). We concluded two things: 1) Tools are required that guide users in writing notification specifications. 2) Tutorials and reference examples are needed that help users to get started quickly by just customizing the examples.

Making the language simple for simple specifications, but expressive enough to support complex notification scenarios was challenging because of usability issues. The more functionality we put in the specification, the more difficult it was to read. We therefore provide only a limited set of functionality that is sufficient for the most common notification scenarios. More complex scenarios can still be realized, but require writing the rules in the native format of the rule engine. This decision also simplified the compilation of the specification into the native rule format.

Currently no escalation rules can be defined in the notification specification language: Eg. if a notification cannot be delivered within a specific time to the receiver, it should escalate to another receiver. It is a functionality that has been asked for and that we plan to provide in future versions.

6.5 Discussion of the Expected Benefits and Costs

Figure 6.3 illustrates the expected impact of NOTICON compared to informal communication (eg. via phone) and conventional notification systems (eg. email per change). The benefit of NOTICON is visualized along three axis:

- The *interruption* describes how much users are interrupted by the arrival of a notification (regardless of whether they need it or do not need it). As shown in the TCO analysis (see section 6.1), the interruption costs of NOTICON are significantly lower than that of an email solution. The highest interruption costs are assumed for informal communication.
- The *number of false negatives* refers to the notifications that should have been communicated but were not. With just informal communication the probability that important information is not communicated to all interested persons

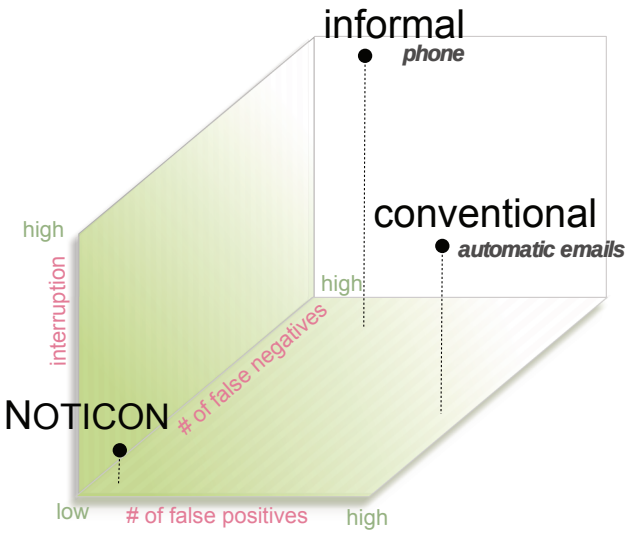

Figure 6.3: Expected Benefit of NOTICON

is very high. Conventional notification technologies can perform significantly better, but face the problem that due to the possibly large number of irrelevant information, users are more likely to miss relevant ones [17]. The powerful rule system of NOTICON can ensure that the right information is delivered to the right persons at the right time.

- The *number of false positives* refers to the notification that were delivered, but are of no interest for the receivers. Conventional notification systems perform worst because they typically broadcast events to a predefined set of receivers, although it is only relevant for a small number of developers. NOTICON drastically reduces the number of false positives because the receivers can be selected based on complex rules that can take the users'- and project-context into account. Informal communication also has a rather low number of false positives; mainly because only a very limited number of people can be reached by this means.

The benefits of NOTICON are much higher than that of other solutions, as are its initial costs: These are mainly caused by 1) the time it takes to install and configure the system and setup the sensors correctly, and 2) the learning costs.

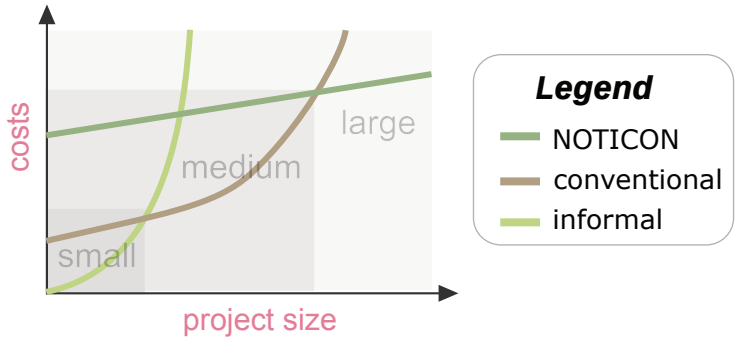

Figure 6.4: Estimated Cost Curves of Notification Solutions

Figure 6.4 illustrates the estimated cost curves of informal communication, conventional notification solutions, and NOTICON :

- It shows that *informal communication* is only feasibly for small projects and that the costs grow exponentially with the project size. As soon as people are not working together in different locations, the likelihood that important project events are not communicated to all concerned persons is very high.

- *Conventional notification* techniques perform very well for small and medium size projects where the amount of notification that needs to be delivered every day is relatively small. But as we showed in section 6.1, the bad ratio between positives and false positives leads to high running costs that can exceed the benefits.

- NOTICON has high initial costs compared to the other approaches which makes it infeasible for small projects. Medium to large size projects can benefit from advanced notification specifications that improves the ratio between positives and false positives. Because there are no constraints on the complexity of the rules, the costs are expected to increase only linear with the project size.

The conclusion is that NOTICON is currently most feasibly in large project, but we think that the installation costs can be decreased significantly if we develop an advanced installation tool that guides administrators through the installation and configuration of the system. This would allow NOTICON to compete with conventional notification techniques in medium sized projects.

7 Summary & Future Work

This chapter summarizes the work and gives and gives a brief outlook on our future plans. It finishes with the "Takeaway Messages" section that condenses our work into three sentences.

Summary

Global Software Development (GSD) projects are complex projects where multiple distributed teams and companies work together in a socio-technical environment to deliver complex software to its clients. They exhibit the following characteristics (see section 2.1):

- They consist of a massive amount of different heterogeneous elements like persons, source code, hardware, software, etc.
- There are many dependencies between the elements; some of them are known, some not.
- The environment (the elements and their dependencies) changes constantly.

These characteristics and the increased demand for agility require an efficient and steady communication between the project members, but which gets hindered by geographical distance, different timezones, different languages, different cultures, different technical infrastructures, etc.

Communicating project events (eg. the change of a requirement) is one aspect of communication in software development projects which is often done informally in small projects. This does not scale well to GSD projects because project members might not know all persons who are impacted by an event, and the informal communication channels (eg. meetings at the coffee machine) are only available at best within one location. The risk that important events are communicated only with big delays or not at all is therefore very high. This can lead to increased costs and delays of the project, and decreases the project members' satisfaction.

Notification functions in tools are a common way to mitigate the risks of information loss and delay (see section 2.3). Users subscribe in tools like SVN or Track to events they are interested in and get an email when such an event happens. In GSD projects this approach is often unfeasible because 1) people might not have access rights for some tools, 2) might not be aware of the tools that are used by other teams, and 3) will receive too much information that is not relevant for them but interrupts their primary tasks. A more efficient context-aware solution is required which provides users with just the information that is currently relevant for them.

We implemented a open source tool prototype called NOTICON. It is a context-aware notification system that uses context information (eg. the activity a user is currently performing, dependencies between components, etc.) to efficiently deliver the right information, at the right time, to the right receipts, without interrupting their current activity (see section 2.3). NOTICON is able to:

- Monitor the current project context across tools to detect situations that are relevant for some users and notify them accordingly. This reduces the amount of notifications a user receives, but increases their importance.
- Display the notifications in the tool that is currently used by a user. This minimizes interruptions caused by notifications, but increases notification awareness.
- Adapt the content of the notification based on the projects context and user preferences. This ensures that all relevant information is included in the notification.

To be able to deliver this functionality NOTICON must integrate various external tools like SVN, Track, RequisitePro, Eclipse, etc. to sense relevant events. We follow best practices from the field of *Enterprise Application Integration* as described in section 2.5.

NOTICON uses a rule engine to allow the specification of advanced notification rules. The rule engine maintains a so called *working memory* that contains the project context (see section 2.4). The project context consists of all relevant project elements (eg. requirements, users, etc.) and their relations. Events that are sensed in external systems lead to changes of the context. Notification rules specify states of the project context that trigger notifications when they occur.

There is a vast amount of functionality that could be packed into a notification system and also the possibility for the systems architecture are numerous. We therefore performed a detailed stakeholder analysis aiming to better understand the stakeholders interests (see section 3.3). The most important requirements we discovered were (section 3.4):

- The system must be built upon standard technologies. This is especially important for future development of NOTICONbecause it attracts new developers who like to use technologies they are already familiar with.
- The system must support the users' work without interfering with their primary tasks. This is important for the acceptance of the system and also has a high impact on its perceived and actual benefit.
- False positives and false negatives must be minimized. False positives are notifications that are delivered to users but are not relevant for them. They just interrupt their primary activity and bring no benefit. False negatives refer to information that should have been delivered to users, but was not.
- The definition of complex notification rules must be supported. It must be possible to define the rules being agnostic of the tools that provide the relevant data. The definition must allow to specify whom, how, when, in which context, due to which event, and with which content a notification should be sent and

what to do in case of an error.

- Users must be able to specify the notification requirements during runtime. This is important to minimize the work of administrators and because rules are expected to change during the project lifetime.

Based on these requirements we developed a prototype in Java that uses various well known open source technologies like ActiveMQ, Mule, and Drools (see chapter 4).

An important requirement is that users must be able to specify the notifications themselves. We therefor developed a domain specific language called *Notification Specification Language* (NSL) that allows to precisely define the rules when a notification should be created. It has been designed to be easy to learn and use by project members, but to be powerful enough to specify complex notification rules.

A notification specification consists of up to eight parts (see section 4.5):

- *Title:* Defines that title of the specification.
- *Receiver:* Specifies who should receive the notifications and whether receivers have to subscribe to get the notifications or need to unsubscribe if they do not want to get them.
- *Context:* The context defines precisely when the notification should be created.
- *Delivery:* Defines whether the notification should be delivered immediately after creation, as batch at a specific time, or depending on context conditions.
- *Channel:* Defines how the notification should be delivered; eg. via email or via the tool that the receiver is currently using.
- *Batch Subject:* In case the notifications are sent in a batch, a subject can be defined for the whole batch.
- *Subject:* Defines the subject of a single notification.

- *Content:* Defines the content of a notification. Any information from the project context can be included.

To gain full benefit of a notification solution the project environment needs to be analyzed first. We proposed an iterative process that can be followed by project managers to 1) discover communication risks, and 2) decide if these risks can be efficiently mitigated with a notification solution like NOTICON . It consists of four steps (see section 3.5):

1. The first step is to find out, if there are high communication risks in a particular project that should be mitigated. We defined a list of risk drivers that help project managers to discover concrete communication risks that can be mitigated with a notification solution.

2. In the second step the project is described as a socio-technical system from four viewpoints: 1) The *People Viewpoint* describes the project organization and the distribution of the project teams. 2) The *Information Viewpoint* describes concepts (eg. requirements) and their relations and dependencies between each other. 3) The *Technology Viewpoint* focuses on the technologies that are used in the project. 3) The *Process Viewpoint* relates the elements from the other viewpoints in the processes that are performed during the project.

3. Based on the diagrams in step 2, the key communications (communications that have a high impact if they do not occur) that can be automated with a notification solution are systematically searched and described.

4. The last step is to describe the notifications and the expected impacts both formally (with the NSL) and informally.

The process and the prototype were evaluated based on a realistic case study that was prepared in cooperation with experts from Siemens PSE (see chapter 5). It showed that it can be performed quickly and provide a solid foundation for discovering communication risks that can be mitigated with notification solutions.

The benefit of our solution was evaluated with a total cost of ownership analysis (see section 6.1); the setting was deduced from the case study. We compared NOTICON

with conventional email notifications with regards to communicating requirement changes and calculated with 70 developers and 30 requirement changes per day for a period of 50 days. Results were that, although the initial investment for NOTICON is significantly higher than that of the email alternative, the notification costs can be reduced by more than 50% with NOTICON .

7.1 Future Work

During the work many ideas were generated on how the system can be improved in the future:

- The **prototype** should be set up as an open source project on Sourceforge [1] so that other developers can participate. We also plan to lower the entrance hurdles for using NOTICON by developing ready to use installation packages that just need to be executed and configured by administrators. Also the user interface should be improved: users should be able to fine tune notifications from within the tools they are currently using. A query functionality is planned that should allow users to retrieve further information after the receipt of a notification.

- The next step for the **process** is a critical evaluation in a real project. We also plan to create a guidance package with the *eclipse process framework* [2] that should help project managers to apply the process in their projects.

- The **notification specification language** will be extended with the possibility to define escalation rules (what should happen if a notification cannot be delivered to a receiver). An editor is planned that helps users in writing specifications.

[1] http://www.sourceforge.net
[2] http://www.eclipse.org/epf

7.2 Takeaway Messages

The three key messages of this work are:

1. GSD projects face high communication risks and some of the risks can be mitigated with notification systems.

2. Existing solutions are inapplicable because they generate either too much or too less notifications which causes high costs in GSD projects.

3. Our solution can be used to systematically describe and discover the notification requirements and can effectively and efficiently deliver 1) the right information, 2) at the right time, 3) to the right persons, without interrupting their current activities.

Bibliography

[1] *Manifesto for Agile Software Development.* http://www.agilemanifesto.org/, Abruf: 29.09.2007

[2] OMG: Notification Service Specification. Version: 2004. http://www.omg.org/technology/documents/formal/notification_service.htm. 2004 (v. 1.1). – Forschungsbericht

[3] *Software Engineering Body of Knowledge.* Endorsed by ACM and IEEE. http://www.swebok.org/. Version: 2004

[4] *TestDirector Requirements Synchronizer for Rational RequisitePro Add-in Guide, Version 8.0.* Version: 2004. http://updates.merc-int.com/testdirector/td80/sync/reqpro/TDSync4RP.pdf, Abruf: 22.09.2007. Mercury Interactive Corporation

[5] Bundesrepublik Deutschland: V-Modell XT, Version 1.2.0. Version: 2004. http://v-modell.iabg.dd. 2004. – Process Model

[6] *Mule Architecture Overview.* Version: 2007. http://mule.mulesource.org/display/MULE/Architecture+Overview, Abruf: 09.10.2007

[7] Abowd, Gregory D. ; Dey, Anind K. ; Brown, Peter J. ; Davies, Nigel ; Smith, Mark ; Steggles, Pete: Towards a Better Understanding of Context and Context-Awareness. In: *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing.* London, UK : Springer-Verlag, 1999. – ISBN 3–540–66550–1, S. 304–307

[8] Addison, Tom ; Vallabh, Seema: Controlling software project risks: an

empirical study of methods used by experienced project managers. In: *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*. Republic of South Africa : South African Institute for Computer Scientists and Information Technologists, 2002. – ISBN 1–58113–596–3, S. 128–140

[9] Baldauf, Matthias ; Dustdar, Schahram ; Rosenberg, Florian: A Survey on Context-Aware systems. In: *International Journal of Ad Hoc and Ubiquitous Computing, forthcoming* (2004). citeseer.ist.psu.edu/baldauf04survey.html

[10] Blaschke, Steffen: Modeling the Improbability of Communication. In: *NAACSOS Conference*. Notre Dame, Indiana, USA, June 2005

[11] Boehm, B. ; Ross, R.: Theory-W software project management: a case study. In: *ICSE '88: Proceedings of the 10th international conference on Software engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1988. – ISBN 0–89791–258–6, S. 30–40

[12] Booker, John E. ; Chewar, C. M. ; McCrickard, D. S.: Usability testing of notification interfaces: are we focused on the best metrics? In: *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–870–9, S. 128–133

[13] Carzaniga, Antonio ; Rosenblum, David S. ; Wolf, Alexander L.: Design and evaluation of a wide-area event notification service. In: *ACM Trans. Comput. Syst.* 19 (2001), Nr. 3, S. 332–383. http://dx.doi.org/http://doi.acm.org/10.1145/380749.380767. – DOI http://doi.acm.org/10.1145/380749.380767. – ISSN 0734–2071

[14] Channabasavaiah, Kishore ; Holley, Kerrie ; Edward Tuggle, Jr.: Migrating to a service-oriented architecture, Part 1. In: *IBM developerWorks* (2003), Dec. http://www.ibm.com/developerworks/library/ws-migratesoa/

[15] Chinnici, Roberto ; Moreau, Jean-Jacques ; Ryman, Arthur ; Weerawarana, Sanjiva: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language / W3C. Version: 2007. http://www.w3.org/TR/wsdl20/. 2007. – W3C Recommendation

[16] Conway, M.E.: How Do Committees Invent? In: *Datamation* 14 (1968), Nr. 4, S. 28–31

[17] Damian, D. ; Izquierdo, L. ; Singer, J. ; Kwan, I.: Awareness in the Wild: Why Communication Breakdowns Occur. In: *Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on* (2007), S. 81–90

[18] Dietz, Jan L. G.: The atoms, molecules and fibers of organizations. In: *Data Knowl. Eng.* 47 (2003), Nr. 3, S. 301–325. http://dx.doi.org/http://dx.doi.org/10.1016/S0169-023X(03)00062-4. – DOI http://dx.doi.org/10.1016/S0169–023X(03)00062–4. – ISSN 0169–023X

[19] Dourish, Paul ; Bly, Sara: Portholes: supporting awareness in a distributed work group. In: *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM Press, 1992. – ISBN 0–89791–513–5, S. 541–547

[20] Dustdar, Schahram ; Gall, Harald ; Hauswirth, Manfred: *Software-Architekturen für Verteilte Systeme*. Springer-Verlag Berlin Heidelberg, 2003

[21] Fabian, Alain ; Felton, David ; Grant, Melissa ; Montabert, Cyril ; Pious, Kevin ; Rashidi, Nima ; Anderson Ray Tarpley, III ; Taylor, Nicholas ; Chewar, C. M. ; McCrickard, D. S.: Designing the claims reuse library: validating classification methods for notification systems. In: *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–870–9, S. 357–362

[22] Fallside, David C. ; Walmsley, Priscilla: XML Schema Part 0: Primer Second Edition / W3C. Version: 2004. http://www.w3.org/TR/xmlschema-0/. 2004. – W3C Recommendation

[23] Filho, Roberto S. S. ; Redmiles, David F.: Striving for versatility in publish/subscribe infrastructures. In: *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*. New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–204–4, S. 17–24

[24] Fischer, Gerhard: User Modeling in Human - Computer Interaction. In: *User Modeling and User-Adapted Interaction* 11 (2001), Nr. 1-2, S. 65–86. – ISSN 0924–1868

[25] Fonseca, Sebastiao B. ; Souza, Cleidson R. B. ; Redmiles, David F.: Exploring the Relationship between Dependencies and Coordination to Support Global Software Development Projects. In: *icgse* 0 (2006), S. 243–. http://dx.doi.org/http://doi.ieeecomputersociety.org/. – DOI http://doi.ieeecomputersociety.org/. ISBN 0–7695–2663–2

[26] Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. In: *Ieee Computer Society Reprint Collection* (1991), S. 324–341

[27] Fowler, Martin: *Patterns of Enterprise Application Architecture*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0321127420

[28] Freimut, Bernd ; Hartkopf, Susanne ; Kaiser, Peter ; Kontio, Jyrki ; Kobitzsch, Werner: An industrial case study of implementing software risk management. In: *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA : ACM Press, 2001. – ISBN 1–58113–390–1, S. 277–287

[29] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[30] Geurts, Gerke ; Geelhoed, Adrie: Business Process Decomposition and Service Identification Using Communication Patterns. In: *The Architecture*

Journal (2004), Jan, Nr. 1

[31] Glinz, Martin ; Wieringa, Roel J.: Guest Editors' Introduction: Stakeholders in Requirements Engineering. In: *IEEE Software* 24 (2007), Nr. 2, S. 18–20. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/MS.2007.42. – DOI http://doi.ieeecomputersociety.org/10.1109/MS.2007.42. – ISSN 0740–7459

[32] Goldhaber, Gerald M. ; Rogers, Donald P.: *Auditing organizational communication systems: the ICA communication audit.* Dubuque, Iowa : Kendall/Hunt Publ., 1979

[33] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *The Java ™Language Specification, Third Edition.* 3rd. Prentice Hall, 2005

[34] Gruber, R.E. ; Krishnamurthy, B. ; Panagos, E.: The architecture of the READY event notification service. In: *Electronic Commerce and Web-based Applications/Middleware, 1999. Proceedings. 19th IEEE International Conference on Distributed Computing Systems Workshops on*, 1999, S. 108–113

[35] Hapner, Mark ; Burridge, Rich ; Sharma, Rahul ; Fialli, Joseph ; Stout, Kate: Java™Message Service Specification / SunMicrosystems, Inc. 2002 (1.1). – Specification

[36] Hargie, Owen ; Tourish, Dennis: *Handbook of Communication Audits for Organisations.* Routledge, 2000

[37] Henricksen, K. ; Indulska, J.: Developing context-aware pervasive computing applications: Models and approach. In: *Pervasive and Mobile Computing* 2 (2006), Nr. 1, S. 37–64

[38] Henricksen, Karen ; Indulska, Jadwiga ; Rakotonirainy, Andry: Modeling Context Information in Pervasive Computing Systems. In: *Pervasive '02: Proceedings of the First International Conference on Pervasive Computing.* London, UK : Springer-Verlag, 2002. – ISBN 3–540–44060–7, S. 167–180

[39] Herbsleb, James D.: Global Software Engineering: The Future of Socio-

technical Coordination. In: *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2829–5, S. 188–198

[40] Hohpe, Gregor: Programmieren ohne Stack: ereignis-getriebene Architekturen. In: *OBJEKTspektrum* (2006), Februrary. http://www.enterpriseintegrationpatterns.com/docs/EDA.pdf. – source is the unedited english version

[41] Hohpe, Gregor ; Woolf, Bobby: *Enterprise Integration Patterns - Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004

[42] Jordon, Diane ; Evdemon, John: Web Services Business Process Execuition Language Version 2.0 / OASIS. Version: April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. 2007. – OASIS Standard

[43] Kantor, M. ; Redmiles, D.: Creating an Infrastructure for Ubiquitous Awareness. In: *Eighth IFIP TC* 13 (2001), S. 431–438

[44] Keller, Kurt: Socio-technical systems and self-organization. In: *SIGOIS Bull.* 17 (1996), Nr. 1, S. 6–7. http://dx.doi.org/http://doi.acm.org/10.1145/236410.236417. – DOI http://doi.acm.org/10.1145/236410.236417. – ISSN 0894–0819

[45] Kirwin, Bill: *Gartner Total Cost of Ownership*. Version: 1987. http://amt.gartner.com/TCO/MoreAboutTCO.htm, Abruf: 29.08.2007. Online

[46] Kontio, Jyrki: The Riskit Method for Software Risk Management, version 1.00 - CS-TR-3782 / UMIACS-TR-97- 38 / University of Maryland. College Park, MD, 1997. – Computer Science Technical Reports

[47] Kontio, Jyrki: Risk management in software development: a technology overview and the Riskit method. In: *ICSE '99: Proceedings of the 21st international conference on Software engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1999. – ISBN 1–58113–074–0, S. 679–680

[48] Kontio, Jyrki ; Getto, Gerhard ; Landes, Dieter: Experiences in improving risk management processes using the concepts of the Riskit method. In: *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA : ACM Press, 1998. – ISBN 1–58113–108–9, S. 163–174

[49] Kroll, Per ; Royce, Walker: Key principles for business-driven development. In: *IBM developerWorks* (2005), Oct. http://www-128.ibm.com/developerworks/rational/library/oct05/kroll/

[50] Laplante, Phillip A. ; Neill, Colin J.: Opinion: The Demise of the Waterfall Model Is Imminent. In: *Queue* 1 (2004), Nr. 10, S. 10–15. http://dx.doi.org/http://doi.acm.org/10.1145/971564.971573. – DOI http://doi.acm.org/10.1145/971564.971573. – ISSN 1542–7730

[51] Lenz, Gunther ; Wienands, Christoph ; Sumser, Jim (Hrsg.): *Practical Software Factories in .NET*. New York : Springer-Verlag, 2006

[52] Loughman, T.P. ; Fleck, R. ; Snipes, R.: A cross-disciplinary model for improved information systems analysis. In: *Industrial Management and Data Systems* 100 (2000), Nr. 8, S. 359–369

[53] Luckham, David: *The Power of Events - An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wsley, 2005

[54] Luhmann, N.: *Social Systems*. Stanford, CA : Stanford University Press, 1995

[55] MacKenzie, C. M. ; Laskey, Ken ; McCabe, Francis ; Brown, Peter F. ; Metz, Rebekah: Reference Model for Service Oriented Architecture 1.0 / OASIS. Version: August 2006. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm. 2006. – Committee Specification

[56] Mark, Gloria ; Gonzalez, Victor M. ; Harris, Justin: No task left behind?: examining the nature of fragmented work. In: *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM Press, 2005. – ISBN 1–58113–998–5, S. 321–330

[57] McCrickard, D. S. ; Chewar, C. M.: Attuning notification design to user goals and attention costs. In: *Commun. ACM* 46 (2003), Nr. 3, S. 67–72. http://dx.doi.org/http://doi.acm.org/10.1145/636772.636800. – DOI http://doi.acm.org/10.1145/636772.636800. – ISSN 0001–0782

[58] McGuinness, Dborah L. ; Harmelen, Frank van: OWL Web Ontology Language Overview / W3C. Version: Feb 2004. http://www.w3.org/TR/owl-features/. 2004. – W3C Recommondation. – last access: 20.08.2007

[59] Moore, James F.: Predators and prey: a new ecology of competition. (1999), S. 121–141. ISBN 0–87584–911–3

[60] Paasivaara, M.: Communication Needs, Practices and Supporting Structures in Global Inter-Organizational Software Development Projects. In: *Workshop on Global Software Development, part of the International Conference on Software Engineering (ICSE)* (2003). http://www.gsd2003.cs.uvic.ca/upload/Paasivaara.pdf

[61] Padayachee, Keshnee: An interpretive study of software risk management perspectives. In: *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*. Republic of South Africa : South African Institute for Computer Scientists and Information Technologists, 2002. – ISBN 1–58113–596–3, S. 118–127

[62] Palmer, Mark: Event Stream Processing - A New Physics of Software. In: *DM Direct* (2005). http://www.dmreview.com/editorial/dmreview/print_action.cfm?articleId=1033537

[63] Panagos, E. ; Rabinovich, M.: Escalations in workflow management systems. In: *CIKM '96: Proceedings of the workshop on on Databases*. New York, NY, USA : ACM Press, 1997. – ISBN 0–89791–948–3, S. 25–28

[64] Papazoglou, Mike ; Heuvel, Willem-Jan van d.: Service oriented architectures: approaches, technologies and research issues. In: *The VLDB Journal The International Journal on Very Large Data Bases* 16 (2007), Juli, Nr. 3,

389–415. http://dx.doi.org/10.1007/s00778-007-0044-3

[65] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. In: *Commun. ACM* 15 (1972), Nr. 12, S. 1053–1058. http://dx.doi.org/http://doi.acm.org/10.1145/361598.361623. – DOI http://doi.acm.org/10.1145/361598.361623. – ISSN 0001–0782

[66] Pietzuch, Peter R. ; Bacon, Jean: Hermes: A Distributed Event-Based Middleware Architecture. In: *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*. Washington, DC, USA : IEEE Computer Society, 2002. – ISBN 0–7695–1588–6, S. 611–618

[67] Preiss, O. ; Wegmann, A.: Stakeholder discovery and classification based on systems science principles. In: *Quality Software, 2001. Proceedings.Second Asia-Pacific Conference on*, 2001, S. 194–198

[68] Ripley, Roger M. ; Yasui, Ryan Y. ; Sarma, Anita ; Hoek, André van d.: Workspace awareness in application development. In: *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA : ACM Press, 2004, S. 17–21

[69] Rising, L. ; Janoff, N.S.: The Scrum software development process for small teams. In: *Software, IEEE* 17 (2000), Nr. 4, S. 26–32. – ISSN 0740–7459

[70] Rosenblum, David S. ; Wolf, Alexander L.: A design framework for Internet-scale event observation and notification. In: *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA : Springer-Verlag New York, Inc., 1997. – ISBN 3–540–63531–9, S. 344–360

[71] Sarma, A. ; Hoek, A. van d.: Towards Awareness in the Large. In: *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE'06)-Volume 00* (2006), S. 127–131

[72] Sarma, Anita ; Noroozi, Zahra ; Hoek, André van d.: Palantir: raising awareness among configuration management workspaces. (2003), S. 444–454.

ISBN 0–7695–1877–X

[73] Satyanarayanan, M.: Coping with uncertainty. In: *Pervasive Computing, IEEE* 2 (2003), Nr. 3, S. 2–2

[74] Schiefer, Josef ; Rozsnyai, Szabolcs ; Rauscher, Christian ; Saurer, Gerd: Event-driven rules for sensing and responding to business situations. (2007), S. 198–205. http://dx.doi.org/http://doi.acm.org/10.1145/1266894.1266934. – DOI http://doi.acm.org/10.1145/1266894.1266934. ISBN 978–1–59593–665–3

[75] Segall, B. ; Arnold, D. ; Boot, J. ; Henderson, M. ; Phelps, T.: Content Based Routing with Elvin4. In: *Proceedings AUUG2K, Canberra, Australia, June* (2000)

[76] Sen, Shilad ; Geyer, Werner ; Muller, Michael ; Moore, Marty ; Brownholtz, Beth ; Wilcox, Eric ; Millen, David R.: FeedMe: a collaborative alert filtering system. (2006), S. 89–98. http://dx.doi.org/http://doi.acm.org/10.1145/1180875.1180890. – DOI http://doi.acm.org/10.1145/1180875.1180890. ISBN 1–59593–249–6

[77] Sena, James A. ; Shani, A. B. (.: Intelligence systems: a sociotechnical systems perspective. In: *SIGCPR '99: Proceedings of the 1999 ACM SIGCPR conference on Computer personnel research*. New York, NY, USA : ACM Press, 1999. – ISBN 1–58113–063–5, S. 86–93

[78] Shen, Haifeng ; Sun, Chengzheng: Flexible notification for collaborative systems. In: *CSCW '02: Proceedings of the 2002 ACM conference on Computer supported cooperative work*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–560–2, S. 77–86

[79] Speier, C. ; Valacich, J.S. ; Vessey, I.: The effects of task interruption and information presentation on individual decision making. In: *Proceedings of the eighteenth international conference on Information systems* (1997), S. 21–36

[80] Spira, JB ; Feintuch, JB: *The Cost of Not Paying Attention: How Interrup-*

tions Impact Knowledge Worker Productivity. 2005

[81] Strang, Thomas ; Linnhoff-Popien, Claudia: A Context Modeling Survey. In: *First International Workshop on Advanced Context Modelling, Reasoning And Management at UbiComp 2004, Nottingham, England, September 7, 2004*, 2004

[82] Ten-Hove, Ron ; Walker, Peter: Java™Business Integration (JBI) 1.0 / Sun Microsystems, Inc. 2005 (JSR 208). – Specification

[83] Truong, Binh A. ; Lee, Young-Koo ; Lee, Sung-Young: Modeling and reasoning about uncertainty in context-aware systems. In: *e-Business Engineering, 2005. ICEBE 2005. IEEE International Conference on*, 2005, S. 102–109

[84] Wahyudin, Dindin ; Heindl, Matthias ; Berger, Ronald ; Schatten, Alexander ; Biffl, Stefan: In-Time Project Status Notification for All Team Members in Global Software Development as Part of Their Work Environments. In: *International Conference on Global Software Engineering (ICGSE 2007)*, 2007

[85] Wahyudin, Dindin ; Heindl, Matthias ; Eckhard, Benedikt ; Schatten, Alexander ; Biffl, Stefan: In-time role-specific notification as formal means to balance agile practices in global software development settings. In: *2nd IFIP Central and East European Conference on Software Engineering Techniques CEE-SET 2007*, 2007

[86] Weiser, Mark: The Computer for the 21st Century. In: *Scientific American* (1991), 02/1991. http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html

[87] Welsh, Matt ; Culler, David ; Brewer, Eric: SEDA: an architecture for well-conditioned, scalable internet services. (2001), S. 230–243. http://dx.doi.org/http://doi.acm.org/10.1145/502034.502057. – DOI http://doi.acm.org/10.1145/502034.502057. ISBN 1–58113–389–8

[88] Whitworth, B.: Social-technical Systems. In: Ghoui, Claude (Hrsg.): *En-*

cyclopedia of Human Computer Interaction. Hershey : Idea Group Reference, 2006, S. 533–541

[89] Wiredu, Gamel O.: A framework for the analysis of coordination in global software development. In: *GSD '06: Proceedings of the 2006 international workshop on Global software development for the practitioner*. New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–404–9, S. 38–44

[90] Zimmer, Tobias: Towards a Better Understanding of Context Attributes. In: *PERCOMW '04: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2106–1, S. 23

List of Figures

2.1 Overview of a GSD project 8
2.2 Project type classification [60] 10
2.3 Unawareness of indirect dependencies can lead to wrong assumptions on the impact of a change. 13
2.4 Example user context for notifications in GSD projects 18
2.5 Architecture layers for context-aware systems [9] 21
2.6 Context classification for the validity of data 24
2.7 Publish-Subscribe Notification System 27
2.8 Context-aware Notification System 28
2.9 Simple event stream processing. 34
2.10 Components of the Drools Rule Engine. 36
2.11 Point to Point Integration vs. Integration Bus 37
2.12 Structure of a WSDL Service Description [82] 40
2.13 Enterprise Service Bus [64] 41
2.14 Message based integration 42
2.15 Event Cloud 44

3.1 Generic stakeholder classification layout 47
3.2 Stakeholder Classification 49
3.3 Introduction Process 59
3.4 Process to rank risk drivers for further process steps. 61
3.5 Example for the people viewpoint 67
3.6 Example for the information viewpoint. 68
3.7 Example for the technology viewpoint 69
3.8 Processes Viewpoint 71

4.1 Mule Server Components [6] . . . 78
4.2 Core components of NOTICON . . . 79
4.3 Message model of NOTICON that shows the messages and the components that consume or produce them. . . . 82
4.4 Core artifacts layer . . . 85
4.5 Illustration of the merge function . . . 86
4.6 Common Artifacts Layer . . . 88
4.7 Simple grammar described in a graphical notation. . . . 91
4.8 Main production rule . . . 93
4.9 Notification title . . . 93
4.10 Notification receiver . . . 94
4.11 bindings, fillwords and variables . . . 95
4.12 Context definition . . . 96
4.13 Relation constraint . . . 97
4.14 Property constraint . . . 98
4.15 Event constraint . . . 100
4.16 Artifact event constraint . . . 100
4.17 Property event constraint . . . 101
4.18 Delivery options . . . 101
4.19 Conditioned delivery . . . 102
4.20 Delivery channel . . . 103
4.21 Notification content . . . 104
4.22 Notification content . . . 104

5.1 The people viewpoints shows the project organization. . . . 108
5.2 Information viewpoint focused on requirements. . . . 109
5.3 State machine for requirements. . . . 109
5.4 Technology viewpoint. . . . 110
5.5 Process viewpoint showing the requirements implementation process. 111
5.6 Aministration application for NOTICON. . . . 117
5.7 NOTICON embedded in Eclipse and Visual Studio. . . . 118
5.8 Daily status report for customer representatives . . . 120

6.1 Total Cost of Ownership Model . . . 122

6.2 Running Costs 128
6.3 Expected Benefit of NOTICON 138
6.4 Estimated Cost Curves of Notification Solutions 139