Handling of Software Quality Defects in Agile Software Development

JÖRG RECH

Fraunhofer Institute for Experimental Software Engineering (IESE)
Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
joerg.rech@iese.fraunhofer.de, http://www.iese.fraunhofer.de

Proposed chapter for the upcoming book:

Agile Software Development Quality Assurance

Editors

Ioannis Stamelos, Aristotle University of Thessaloniki, Greece

Panagiotis Sfetsos, Technological Educational Institution of Thessaloniki, Greece

Table of Content:

Introduction 1
Background 3
Quality Defects and Quality Defect Discovery 10
Handling of Quality Defects 13
An Annotation Language to Support Quality Defect Handling 18
Summary & Outlook 22
References 23

Handling of Software Quality Defects in Agile Software Development

Abstract: *Software quality assurance is concerned with the efficient and effective development of large, reliable, and high-quality software systems. In agile software development and maintenance, refactoring is an important phase for the continuous improvement of a software system by removing quality defects like code smells. As time is a crucial factor in agile development, not all quality defects can be removed in one refactoring phase (esp. in one iteration). Documentation of quality defects that are found during automated or manual discovery activities (e.g., pair programming) is necessary to avoid waste of time by rediscovering them in later phases. Unfortunately, the documentation and handling of existing quality defects and refactoring activities is a common problem in software maintenance. To recall the rationales why changes were carried out, information has to be extracted from either proprietary documentations or software versioning systems. In this chapter, we describe a process for the recurring and sustainable discovery, handling, and treatment of quality defects in software systems. An annotation language is presented that is used to store information about quality defects found in source code and that represents the defect and treatment history of a part of a software system. The process and annotation language can not only be used to support quality defect discovery processes, but is also applicable in testing and inspection processes.*

Keywords: *software quality assurance, source code annotation, software documentation language, software quality, software refactoring*

Introduction

The success of software organizations – especially those that apply agile methods – depends on their ability to facilitate continuous improvement of their products in order to reduce cost, effort, and time-to-market, but also to restrain the ever increasing complexity and size of software systems. Nowadays, industrial software development is a highly dynamic and complex activity, which is not only determined by the choice of the right technologies and methodologies but also by the knowledge and skills of the people involved. This increases the need for software organizations to develop or rework existing systems with high quality within short periods of time using automated techniques to support developers, testers, and maintainers during their work.

Agile software development methods were invented to minimize the risk of developing low-quality software systems with rigid process-based methods. They impose as little overhead as possible in order to develop software as fast as possible and with continuous feedback from the customers. These methods (and especially extreme programming (XP)) are based upon several core practices, such as *simple design*, meaning that systems should be built as simply as possible and complexity should be removed, if at all possible.

In agile software development, organizations use quality assurance activities like refactoring to tackle defects that reduce software quality. *Refactoring* is necessary to remove *quality defects* (i.e., bad smells in code, architecture smells, anti-patterns, design flaws, negative design characteristics, software anomalies, etc.), which are introduced by quick and often unsystematic development. As time is a crucial factor in agile development, not all quality defects can be removed in one refactoring phase (esp. in one iteration). But the effort for the manual discovery, handling, and treatment of these quality defects results in either incomplete or costly refactoring phases.

A common problem in software maintenance is the lack of documentation to store this knowledge required for carrying out the maintenance tasks. While software systems evolve over time, their transformation is either recorded explicitly in a documentation or implicitly through a versioning system. Typically, problems encountered or decisions made during the development phases get lost and have to be rediscovered in later maintenance phases. Both expected and unexpected CAPP (corrective, adaptive, preventive, or perfective) activities use and produce important information, which is not systematically recorded during the evolution of a system. As a result, maintenance becomes unnecessarily hard and the only countermeasures are, for example, to document every problem, incident, or decision in a documentation system like bugzilla (Serrano & Ciordia, 2005). The direct documentation of quality defects that are found during automated or manual discovery activities (e.g., code analyses, pair programming, or inspections) is necessary to avoid waste of time by rediscovering them in later phases.

In order to support software maintainers in their work, we need a central and persistent point (i.e., across the product's life-cycle) where necessary information is stored. To address this issue, we introduce our annotation language, which can be used to record information about quality characteristics and defects found in source code, and which represents the defect and treatment history of a part of a software system. The annotation language can not only be used to support quality defect discovery processes, but is also applicable for testing and inspection processes. Furthermore, the annotation language can be exploited for tool support, with the tool keeping track and guiding the developer through the maintenance procedure.

Our research is concerned with the development of techniques for the discovery of quality defects as well as a quality-driven and experience-based method for the refactoring of large-scale software systems. The instruments developed consist of a technology and methodology to support decisions of both managers and engineers. This support includes information about where, when, and in what configuration quality defects should be engaged to reach a specific configuration of quality goals (e.g., improve maintainability or reusability). Information from the diagnosis of quality defects supports maintainers in selecting countermeasures and acts as a source for initiating preventive measures (e.g., software inspections).

This chapter targets the handling of quality defects in object-oriented software systems and services. It is concerned with the theory, methodology, and technology for the handling of defects that deteriorate software qualities as defined in ISO 9126 (e.g., maintainability, reusability, or performance). We describe the relevant background and related work concerning quality defects and quality defect handling in agile software projects as well as existing handling techniques and annotation languages. The subsequent section encompasses the morphology of quality defects as well as their discovery techniques. As the core of this paper, we present the techniques for handling quality defects after their discovery in an agile and time-critical environment and define an annotation language to record information about quality defects and their history in source code. Thereafter, a section is used to describe the annotation language that is used to record the treatment history and decisions in the code itself. Finally, we summarize several lessons learned and requirements one should keep in mind when building and using quality defect handling methods and notations in an agile environment. At the end of this chapter, we summarize the described approach and give an outlook to future work and trends.

Background

This section is concerned with the background and related work in agile software engineering, refactoring, and quality defects. It gives an overview of quality defect discovery, the documentation of defects, as well as source code annotation languages.

Agile Software Development

Agile software development methods impose as little overhead as possible in order to develop software as fast as possible and with continuous feedback from the customers. Agile methods have in common that small releases of the software system are developed in short iterations in order to create a running system with a subset of the functionality needed for the customer. Therefore, the development phase is split into several activities, which are followed by small maintenance phases. In contrast to traditional, process-oriented SE, where all requirements and use cases are elicited, agile methods focus on few essential requirements and incrementally develop a functional system in several short development iterations.

Today, Extreme Programming (XP) (Beck, 1999) is the best-known agile software development approach. Figure 1 shows the general process model of XP, which is closely connected to refactoring, basically being its cradle (Beck & Fowler, 1999).

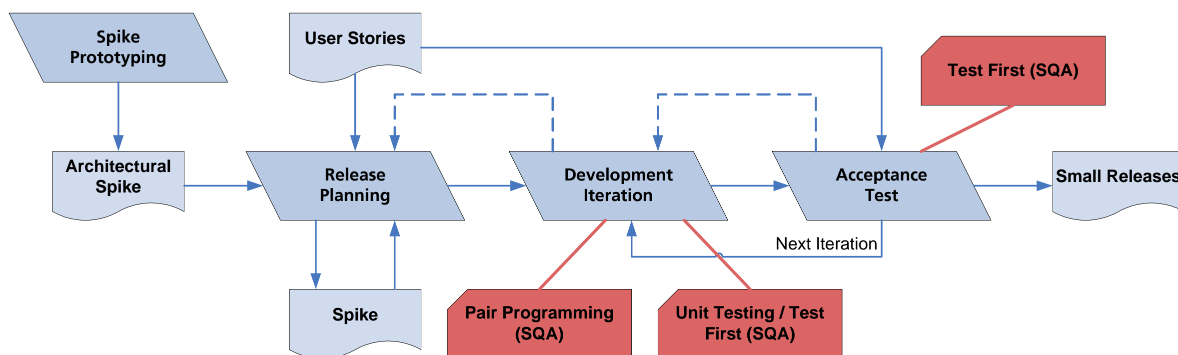

Figure 1. Agile Software Development (here the XP process)

These agile methods (and especially extreme programming (XP)) are based upon 12 principles (Beck, 1999). We mention four of these principles, as they are relevant to our work.

- *The Planning Game* is the collective planning of releases and iterations in the agile development process and is necessary for quickly determining the scope of the next release. If the requirements for the next iteration are coherent and concise, more focus can be given to one topic or subsystem without making changes across the whole system.
- *Small releases* are used to develop a large system by first putting a simple system into production and then releasing new versions in short cycles. The smaller the change to the system, the smaller the risk of introducing complexity or defects that are overlooked in the refactoring (or SQA) phases.

- *Simple design* means that systems are built as simply as possible, and complexity in the software system is removed, if at all possible. The more understandable, analyzable, and changeable a system is, the less functionality has to be refactored or reimplemented in subsequent iterations or maintenance projects.
- *Refactoring* is necessary for removing qualitative defects that are introduced by quick and often unsystematic development. Decision support during refactoring helps the software engineer to improve the system.

In the highly dynamic processes used in agile methods, teams and organizations need automated tools and techniques that support their work without consuming much time. Especially in the refactoring phase, where the software is revised, automation can be used to detect *quality defects* such as code smells (Fowler, 1999), antipatterns (Brown et al., 1998), design flaws (Riel, 1996), design characteristics (Whitmire, 1997), or bug patterns (Allen, 2002). Techniques from KDD support the refactoring of software systems (Rech, 2004), and techniques from knowledge management can foster experience-based refactoring (Rech & Ras, 2004).

Quality Defect Discovery

A central research problem in software maintenance is still the inability to change software easily and quickly (Mens & Tourwe, 2004). To improve the quality of their products, organizations often use quality assurance techniques to tackle defects that reduce software quality. The techniques for the discovery of quality defects are based upon several research fields.

- *Software Inspections* (Aurum et al., 2002; Ciolkowski et al., 2002), and especially code inspections are concerned with the process of manually inspecting software products in order to find potential ambiguities as well as functional and non-functional problems (Brykczynski, 1999). While the specific evaluation of code fragments is probably more precise than automated techniques, the effort for the inspection is higher, the completeness of an inspection regarding the whole system is smaller, and the number of quality defects searched for is smaller.
- *Software Testing* (Liggesmeyer, 2003) and debugging is concerned with the discovery of defects regarding the functionality and reliability as defined in a specification or unit test case in static and dynamic environments.
- *Software product metrics* (Fenton & Neil, 1999) are used in software analysis to measure the complexity, cohesion, coupling, or other characteristics of the software product, which are further analyzed and interpreted to estimate the effort for development or to evaluate the quality of the software product. Tools for software analysis in existence today are used to monitor dynamic or static aspects of software systems in order to manually identify potential problems in the architecture or find sources for negative effects on the quality.

Furthermore, several specific techniques for quality defect discovery already exist (Marinescu, 2004; Rapu et al., 2004). Most of the tools such as Checkstyle, FindBugs, Hammurapi, or PMD analyze the source code of software systems to find violations of project-specific programming guidelines, missing or overcomplicated expressions, as well as potential language-specific functional defects or bug patterns. Nowadays, the Sotograph can identify "architectural smells" that are based on metrics regarding size or coupling (Roock & Lippert, 2005).

But the information from these techniques and the resulting CAPP or *refactoring* activities are typically lost after some time if they are not documented in external documents or *defect management* systems (e.g., bugzilla). And even these external data source are prone to get lost over several years of maintenance and infrastructure changes. The only information that will not get lost is typically the source code itself.

Refactoring

Beside the development of software systems, the effort for software evolution and maintenance is estimated to amount to 50% to 80% of the overall development cost (Verhoef, 2000). One step in the evolution and development of software systems is the process of reworking parts of the software in order to improve its structure and quality (e.g., maintainability, reliability, usability, etc.), but not its functionality. This process of improving the internal quality of object-oriented software systems in agile software development is called *Refactoring* (Fowler, 1999). While refactoring originates in from the agile world, it can, nevertheless, be used in plan-driven (resp. heavyweight) software engineering. In general, refactoring (Fowler, 1999; Mens & Tourwe, 2004) is necessary to remove quality defects that are introduced by quick and often unsystematic development.

The primary goal of agile methods is the rapid development of software systems that are continuously adapted to customer requirements without large process overhead. During the last few years, refactoring has become an important part in agile processes for improving the structure of software systems between development cycles. Refactoring is able to reduce the cost, effort, and time-to-market of software systems. Development, maintenance, and reengineering effort are reduced by restructuring existing software systems (on the basis of best practices, design heuristics, and software engineering principles), especially in the process of understanding (the impact of new changes in) a system. A reduction of effort also reduces the length of projects and therefore, cost and time-to-market. Furthermore, refactoring improves product quality and therefore is able to reduce the complexity and size of software systems. Especially in agile software development, methods as well as tools to support refactoring are becoming more and more important (Mens et al., 2003).

However, performing manual discovery of quality defects that should be refactored result in either very short or costly refactoring phases. While several automations for refactoring have already been developed (e.g., “extract method” refactoring), the location, analysis, and removal is still an unsystematic, intuitive, and manual process. Today, several techniques and methods exist to support software quality assurance (SQA) on higher levels of abstraction (e.g., requirement inspections) or between development iterations (e.g., testing). Organizations use techniques like refactoring to tackle *quality defects* (i.e., bad smells in code (Beck & Fowler, 1999), architecture smells (Roock & Lippert, 2005), anti-patterns (Brown et al., 1998), design flaws (Riel, 1996; Whitmire, 1997), and software anomalies (IEEE-1044, 1995), etc.) that reduce software quality.

But refactoring does not stop after discovery; even if we had solved the problem of discovering every quality defect possible, the information about the defect, the rationales of whether it is removed (or not), and the refactorings used have to be documented in order to support maintainers and reengineers in later phases. If one knows how to remove a specific quality defect or a group of quality defects, one still needs support, as it is not clear where and under which

conditions which refactoring activities should be used. Furthermore, product managers need support to organize chains of refactorings and to analyze the impact of changes due to refactorings on the software system. Analogously, quality managers and engineers need information to assess the software quality, identify potential problems, select feasible countermeasures, and plan the refactoring process as well as preventive measures (e.g., code inspections).

Defect Documentation

Today, various repositories exist for documenting of information about defects, incidents, or other issues regarding software changes. This information can be stored in configuration management systems (e.g., CVS, SourceSafe), code reuse repositories (e.g., ReDiscovery, InQuisiX), or *defect management systems*.

The last category is also known as bug tracking (Serrano & Ciordia, 2005), issue tracking (Johnson & Dubois, 2003), defect tracking (Fukui, 2002), or source code review systems (Remillard, 2005). They enable a software engineer to record information about the location, causes, effects, or reproducibility of a defect. Typical representatives of defect management systems are open-source variants such as Bugzilla (Serrano & Ciordia, 2005), Scarab (Tigris, 2005), Mantis (Mantis, 2005), or TRAC (TRAC, 2005). Commercial versions include Tuppas (Tuppas, 2005), Census from Metaquest (MetaQuest, 2005), JIRA from Atlassian (Atlassian, 2005), or SSM from Force10 (Force10, 2005). These tools are predominantly used in defect handling to describe defects on the lower abstractions of software systems (i.e., source code) (Koru & Tian, 2004) separated from the code.

Defect classification schemes (Freimut, 2001; Pepper et al., 2005) like ODC (Orthogonal Defect Classification) (Chillarege, 1996) are used, for example, in conjunction with these tools to describe the defects and the activity and status a defect is involved in. The ODC process consists of an opening and closing process for defect detection that uses information about the target for further removal activities. Typically, removal activities are executed, but changes, decisions, and experiences are not documented at all – except for small informal comments when the software system is checked into a software repository like CVS.

From our point of view, the direct storage of information about defects, decisions about them, or refactorings applied in the code (as a central point of information) via *annotation languages* such as JavaDoc (Kramer, 1999), doxygen (van Heesch, 2005), or ePyDoc (Loper, 2004) seems to be a more promising solution. The next section describes the relevant background and related work for annotation languages, which are used to record historical information about the evolution of a code fragment (e.g., a method, class, subsystem, etc.).

Source Code Annotation Languages

Annotation languages such as JavaDoc (Kramer, 1999), ePyDoc (ePyDoc, 2005), ProgDOC (Simonis & Weiss, 2003), or Doxygen (van Heesch, 2005) are typically used to describe the characteristics and functionality of code fragments (i.e., classes, methods, packages, etc.) in the source code itself or in additional files. Today several extensions, especially to JavaDoc, are known that enable us to annotate which patterns (Hallum, 2002; Torchiano, 2002), aspects (Sametinger & Riebisch, 2002), or refactorings (Roock & Havenstein, 2002) were or will be used on the source code, and which help us to describe characteristics such as invariants, pre-/

post-conditions, or reviewer names (JSR-260, 2005; Tullmann, 2002). These extensions to the annotation language are called taglets. They are used by doclets in the extraction using, for example, the JavaDoc program. These tools collect the distributed information blocks and generate a (on-line) documentation rendered in HTML or another file format (e.g., PDF) for better viewing. Typically, these documentations describe the application program interface (API) as a reference for software engineers. Similarly, tools and notations like Xdoclet offer additional tags that are used to generate many artifacts such as XML descriptors or source code. These files are generated from templates using the information provided in the source code and its JavaDoc tags.

Typical content of code annotations is, for example, used to describe the:

- Purpose of a class, field, or method.
- Existence of (functional) defects or workarounds.
- Examples of using the code fragment.

In the following sections and tables, we describe the tags currently available for annotating source code using JavaDoc. JavaDoc is a name for an annotation language as well as the name of a tool from Sun Microsystems to generate API documentation and is currently the industry standard for documenting software systems in Java. The tool uses the tags from the JavaDoc language to generate the API documentation in HTML format. It provides an API for creating doclets and taglets, which allows extending the system with one's own tags (via taglets) and the documentation with additional information (via doclets).

As listed in Table 1, JavaDoc currently consists of 19 tags that might be used to describe distinguished information (e.g., such as return values of a method) or to format text passages (e.g., to emphasize exemplary source code). The standard tags appear as "`@tag`" and might include inline tags, which appear within curly brackets "`{@tag}`". Inline tags only appear within, respectively behind, standard tags or in the description field (e.g., "`@pat.name` … `{@pat.role …}`").

Developers can use the JavaDoc tags when documenting source code in a special comment block by starting it with "`/**`" and ending it with "`*/`". A tag is indicated by using an "`@`" ("at") sign right before the tag name. An example of a JavaDoc comment used for a method is:

`/**`	Start of JavaDoc comment
`* Sorts an array using quicksort`	Description of the method
`* @author John Doe`	Indicate the author
`* @param productArray`	Describe a parameter
`* @return Array The sorted array`	Describe the return value
`*/`	End of JavaDoc comment

Table 1 General tags of the JavaDoc annotation language

Tag	Description	Origin	Type
@author	May appear several times and indicates who has created or modified the code.	JavaDoc 1.0	Context
@param	Describes one parameter of a method (or template class).	JavaDoc 1.0	Function
@return	Describes the returned object of a method.	JavaDoc 1.0	Function
@throws	Describes the (exception-) objects that are thrown by this method.	JavaDoc 1.2	Function
@exception	Synonym for @throws	JavaDoc 1.0	Function
@version	States the version of this code structure.	JavaDoc 1.0	Context
@since	States the version since when this code was implemented and available to others.	JavaDoc 1.1	Context
@deprecated	Indicates that this code structure should not be used anymore.	JavaDoc 1.0	Status
@see	Adds a comment or link to the "See also" section of the documentation. May link to another part of the documentation (i.e., code).	JavaDoc 1.0	Reference
@serialData	Comments the types and order of data in a serialized form.	JavaDoc 1.2	Context
@serialField	Comments a ObjectStreamField.	JavaDoc 1.2	Context
@serial	Comments default serializable fields.	JavaDoc 1.2	Context
<@code>	Formats text in code font (similar to <code>).	JavaDoc 1.5	Format
<@docRoot>	Represents the relative path to the root of the documentation.	JavaDoc 1.3	Reference
<@inheritDoc>	Copies the documentation from the nearest inherited code structure.	JavaDoc 1.4	Reference
<@link>	Links to another part of the documentation (i.e., code structure) as the @see tag but stays inline with the text and is formated as "code".	JavaDoc 1.2	Reference
<@linkPlain>	Identical to <@link> but is displayed in normal text format (i.e., not code format)	JavaDoc 1.4	Reference
<@literal>	Displays text without interpreting it as HTML or nested JavaDoc.	JavaDoc 1.5	Format
<@value>	The value of a local static field or of the specified constant in another code.	JavaDoc 1.4	Reference

As an extension to JavaDoc, four refactoring tags were developed in (Roock & Havenstein, 2002) as described in Table 2.

Table 2 Refactoring tags by Roock and Havenstein

Tag	Description
@past	Describes the previous version of the signature.
@future	Describes the future signature of the element.
@paramDef	States the default value expected for a parameter. The syntax is @paramDef <parameter> = <value>
@default	Defines the default implementation of an abstract method.

To note the existence of patterns in a software system as well as the task and role as described in the pattern definitions, several tags were developed by (Torchiano, 2002) and are listed in Table 3.

Table 3 Pattern tags by Torchiano

Tag	Description
@pat.name	States the standard name of the pattern as defined in (Gamma et al., 1994) (and other).

<@pat.role>	Inline-tag of pat.name that describes the part of the pattern that is represented by this element. E.g., “Leaf” in a composite pattern.
@pat.task	Describes the task performed by the pattern or its role.
@pat.use	Describes the use of the pattern or a role, typically by a method.

Furthermore, several other groups of annotations exist for various purposes. The following tags are from (Roock & Havenstein, 2002) (@contract), (Kramer, 1998) (@inv, @pre, @post), (Tullmann, 2002) (@issue, @todo, @reviewedBy, @license), (Sametinger & Riebisch, 2002) (@pattern, @aspect, @trace), and (JSR-260, 2005) (@category, @example, @tutorial, @index, @exclude, @todo, @internal, @obsolete, @threadSafe).

Table 4 Other Tags

Tag	Description
@contract	Defines bounds of a parameter (or other value). Syntax is “@contract <requires> <min> <= <parameter> <= <max>”
@inv, @invariant	States an invariant. Syntax is “@inv <boolean expression>”
@pre	States the precondition for a method.
@post	States the postcondition for a method. This includes information about side effects (e.g., changes to global variables, fields in an object, changes to a parameter, and return values (except if stated in @return).
@issue	Indicates a new requirement or feature that could be implemented. Syntax is @issue [description ...]
@reviewedBy	Indicates a code review for the associated class/interface was completed by a reviewer. Syntax is @reviewedby <name> <date> [notes ...]
@license	Indicates the copyright license used for this code fragment. Syntax is @license [description ...]
@category	Annotates the element with a free attribute / category. @category <category>
@example	@example <description>
@tutorial	Link to a tutorial
@index	Defines the text that should appear in the index created by JavaDoc.
@exclude	States that this element should not be included in the API by the JavaDoc command.
@todo	Indicates that further work has to be done on this element.
@internal	Comments to this element that are internal to the developer or company.
@obsolete	Used if deprecated elements are actually removed from the API.
@threadSafe	Indicates whether this element is threadsafe.
@pattern	Formally describes a pattern existence with the syntax @pattern <pattern name>.<instance name> <role name> <text>
@aspect	Describes an aspect existence with the syntax @aspect <name> <text>
@trace	Describes a pattern existence with the syntax @trace <name> <text>

The characteristics of source code annotation languages can be differentiated by the number of tags and the formality of their expressiveness. We differentiate between three categories of formality:

- Formal: An explicit and unambiguous specification of the content. A formal tag might include an informal section like a description or note to the formal part (e.g., the tag “param”

in JavaDoc has an informal part to describe the meaning of the parameter). In particular, the formal part of a tag must be processable by a computer.

- Semi-formal: A structured or formal representation that is ambiguous or not directly processable by a computer.
- Informal: An unstructured and possibly ambiguous specification of content.

In summary, the tags used in JavaDoc and its extensions can be used to describe characteristics of the source code on a relatively granular or semi-formal level. The processing of these annotations can be used to generate API documentations with additional information about patterns, aspects, or signature changes. The recording of quality defects discovered and refactorings applied as well as rationales or experiences about their application can only be accomplished using free text in the API description.

Furthermore, these annotation languages and their extensions have different target areas in the field of software quality assurance in order to store information about tests, inspections, patterns, and refactorings. Table 5 shows a comparison of several annotation languages in relevant areas:

Table 5 Annotation languages in comparison

Language	Extension	# of Tags	Test Info	Inspection Info	Pattern Info	Refactoring Info
JavaDoc 1.5	Standard	19	No	No	No	No
	Roock & Havenstein	5	Semi-Formal (1)	No	No	Informal (4)
	Torchiano	10	No	No	Semi-Formal (3)	No
	Sametinger & Riebisch	3	No	No	Informal (3)	No
	Tullmann	4	No	Informal (1)	No	No
	Kramer	3	Informal (3)	No	No	No
	JSR-260	9	No	No	No	No

Quality Defects and Quality Defect Discovery

The main concern of software quality assurance (SQA) is the efficient and effective development of large, reliable, and high-quality software systems. In agile software development, organizations use techniques like refactoring to tackle “bad smells in code” (Beck & Fowler, 1999), which reduce software qualities such as maintainability, changeability, or reusability. Other groups of defects that do not attack functional, but rather non-functional aspects of a software system are architecture smells (Roock & Lippert, 2004), anti-patterns (Brown et al., 1998), design flaws (Riel, 1996; Whitmire, 1997), and software anomalies in general (IEEE-1044, 1995).

In this chapter, we use the umbrella term *quality defects* (QD) for any defect in software systems that has an effect on software quality (e.g., as defined in ISO 9126), but does not directly affect functionality. Whether the quality defect is automatically discoverable (Dromey, 1996, 2003) or not (Lauesen & Younessi, 1998) – an annotation language and method that can be used to support the handling of quality defects should record information about quality characteristics and quality defects in order to represent their status and treatment history. This section will

elaborate on this concept and describe several quality defects, their interrelation, symptoms, and effects.

Today, various forms of quality defects exist with different types of granularity. Some target problems in methods and classes, while others describe problems on the architecture or even process levels. In this chapter, we only focus on quality defects on the code level. The representatives on this level are:

- **Code Smells**: The term code smell is an abbreviation of "bad smells in code", which were described in (Beck & Fowler, 1999). Today, we have many code smells that are semi-formally described and can be used for manual inspection and discovery. There are at least 38 known code smells with 22 in (Fowler, 1999), 9 new ones in (Wake, 2003), 5 new ones in (Kerievsky, 2005), and 2 in (Tourwe & Mens, 2003). Code smells are indicators for refactoring and typically include a set of alternative refactoring activities in their description, which might be used to remove them.
- **Architectural Smells**: Very similar to code smells are architectural smells that describe problems on the design level. Yet, the 31 architectural smells described in (Roock & Lippert, 2005) do not only apply on the design level but also on the code level. They typically describe problems regarding classes in object-oriented software and interrelations between them.
- **Anti-patterns**: Design patterns (Gamma et al., 1994) and anti-patterns (Brown et al., 1998) represent typical and reoccurring patterns of good and bad software architectures and were the start of the description of many patterns in diverse software phases and products. While patterns typically state and emphasize a single solution to multiple problems, anti-patterns typically state and emphasize a single problem to multiple solutions. An anti-pattern is a general, proven, and non-beneficial problem (i.e., bad solution) in a software product or process. It strongly classifies the problem that exhibits negative consequences and provides a solution. Built upon similar experiences, these anti-patterns represent "worst-practices" about how to structure or build a software architecture. An example is the "lava flow" anti-pattern, which warns about developing a software system without stopping sometimes and reengineering the system. The larger and older such a software system gets, the more dead code and solidified (bad) decisions it carries along.
- **Bug Patterns**: These patterns are concerned with functional aspects that are typically found in debugging and testing activities. In (Allen, 2002), 15 bug patterns are described, which describe underlying bugs in a software system.
- **Design Flaws & (Negative) Design Characteristics**: Whitmire describes nine distinct and measurable characteristics (Whitmire, 1997) of an object-oriented design. These characteristics such as "similarity" describe the degree to which two or more classes or domain-level abstractions are similar in terms of their structure, function, behavior, or purpose.
- **Design Heuristics**: Design heuristics provide support on how to construct software systems and, in a way, define quality defects by their absence. They range from the "information hiding" principle to guidelines such as "Eliminate irrelevant classes from your design".

There are 61 design heuristics described in (Riel, 1996) and 14 principles described in (Roock & Lippert, 2005).

As listed, there are many quality defects of various granularities and described in different forms. To give a more concrete idea of quality, we describe two of them in the following:

- **Long Method:** In object-oriented programming, one should pay attention to the fact that methods are not too long. The longer a method, the more difficult it is to be understood by a developer. Comprehensibility and readability are negatively affected by the length of a method and thus negatively affect maintainability and testability. Moreover, a short understandable method typically needs less comments than a long one. Another advantage of short methods is the fact that a developer does not constantly scroll and break his reading flow. The most obvious method for discovering long methods is the metric number of lines (LOC) per method. But the question of which method is too long and constitutes a problem is not easily answered. This must either be specified or found by detecting anomalies from the standard distribution. Nevertheless, a method exceeding this threshold value must not necessarily be shortened if other, more important, quality constraints would be negatively affected.
- **Shotgun Surgery:** This denotes the problem that several classes are always changed in the same group, for example, if the system is adapted to a new data base scheme and the same two classes are changed each time. The expandability of the system is thus constrained, and if one class is forgotten during a change, it is more likely to fail. The discovery of shotgun surgery is very difficulty and requires either change metrics or specified rules.

While these problems might not represent problems that have a directly tangible effect on quality, they might become problematic in future evolution or refactoring activities and should be removed as fast as possible – if the time is available. These are only two of many described quality defects. Nevertheless, they show that quality defects describe problems on different levels of complexity and might occur in parallel in one situation (i.e., in one code fragment).

Figure 2 depicts the general model for the concepts that are used to describe quality defects and that are linked to them. A software system might have *predispositions* that foster or enable the creation of quality defects. These defects themselves have *causes* that are responsible for the defects being integrated into the system. The quality *defects* might have a negative as well as a positive effect on specific qualities and are perceivable via specific *symptoms*. Finally, the defects are solved or removed via specific treatments after they are discovered, or the causes might be prevented by special preventive measures.

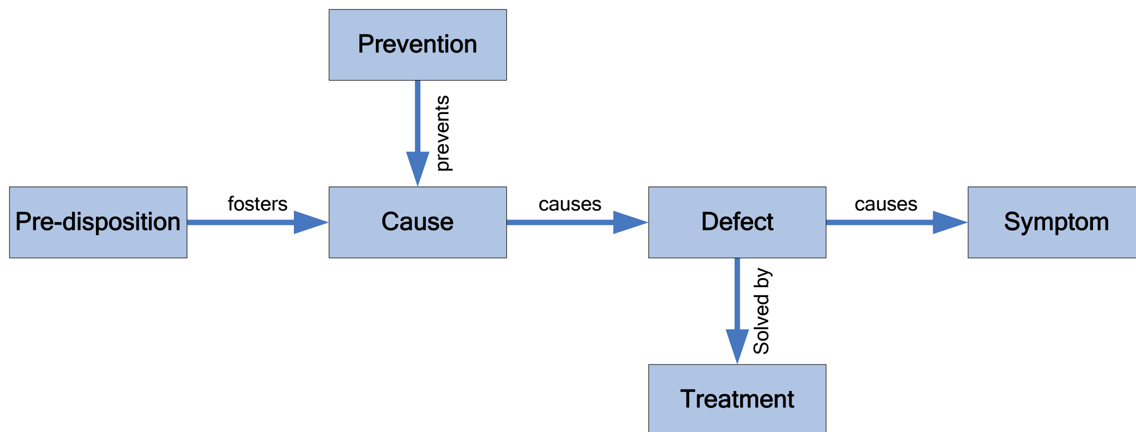

Figure 2. Conceptual model of the quality defect ontology (software product level)

In Software Engineering (SE) and especially in the case of quality defects for a software product, the context of a defect can be described as listed in Table 6.

Table 6 Examples for Software Engineering techniques

	Example 1	**Example 2**
Predisposition	Data processing system	Lack of good architecture / design
Cause	Large data processing algorithms	Distributed functionality
Defect	"Long Method"	"Shotgun surgery"
Side-effects of defect	Increase analyzability effort	Increased maintenance effort
Symptom	Many lines of code	Recurrent changes to the the same units
Treatment	Extract Method	Inline Class(es)
Side-effects of treatment	Increased subroutine calls (worsens performance)	Divergent Change
Prevention	Optimize algorithm (phase)	Pattern-based Architecture

In the example on the right side, the predisposition "bad architecture" causes a "cluttered functionality", which results in a "shotgun surgery" defect. This quality defect can be discovered by the symptom of "recurring changes to the same set of software units" and might be removed by the "Inline Class" refactoring. A prevention of this problem would be a "good" systematic architecture with clear separation of functionality, e.g., in the form of a pattern-based architecture.

Handling of Quality Defects

Typically, during agile development with short iterations, several quality defects are introduced into the software system and are discovered especially in the refactoring phase. To facilitate the annotation of source code and the processing of quality defect removal, a mechanism to store the information from the QDD process is required.

The Handling Process

In the following, the general process of quality defect discovery and refactoring is depicted. Figure 3 shows the process model that is either initiated during software development or during a special maintenance (resp. refactoring) activity.

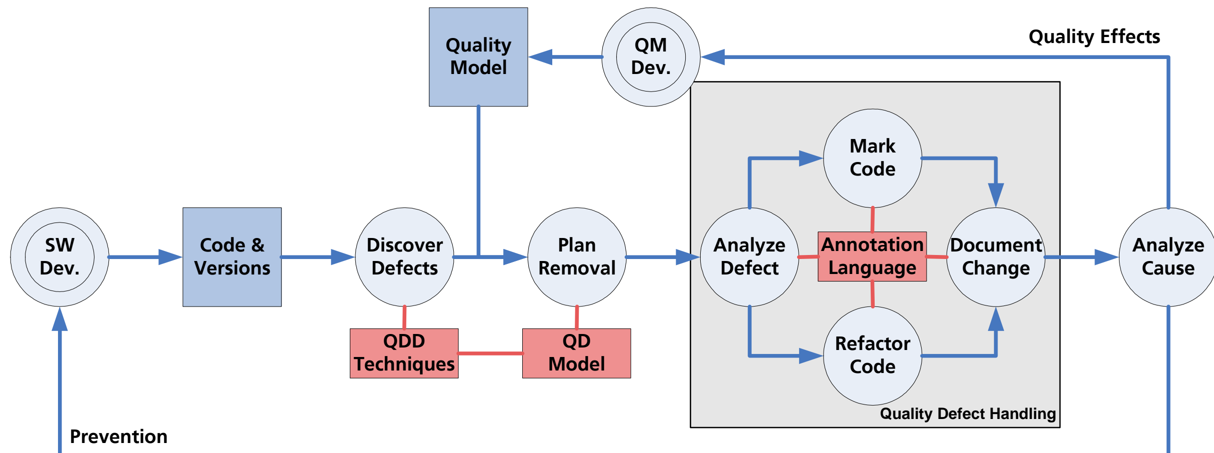

Figure 3. The Quality Defect Discovery and Handling Process Model

In the execution of the process, the following sub-processes are performed:

- *Discover Defects*: Manual or automatic quality defect discovery techniques are used to analyze the source code and versions thereof from the software repository. Potential quality defects are identified and the affected code (of the most current version) is annotated.
- *Plan Removal*: Based on the discovered quality defects (annotated with a special tag) and a previously defined quality model, a sequential plan for the refactoring of the software system (or part thereof) is constructed.
- *Analyze Defects*: The software engineer processes the list of potential quality defects based on their priority, analyzes the affected software system (or part thereof), and decides if the quality defect is truly present and if the software system can be modified without creating too many new quality defects.
- *Refactor Code*: If the quality defect is to be removed from the software system, the engineer is briefed about the existing quality defects and their rationales as well as about available refactorings, their impact on software quality, and previously made experiences with the kind of quality defect and refactoring at hand.
- *Mark Code*: If a potential quality defect is unavoidable or its removal would have a negative impact on an important quality (e.g., performance), this decision is recorded in the affected part of the software system to prevent future analysis of this part.
- *Document Change*: After the refactoring or marking, the software system is annotated with specific tags about the change or decision, and the experience about the activity is recorded within an experience database (i.e., a database in an experience factory (Basili et al., 1994b) for storing, formalizing, and generalizing experiences about software development and

refactoring activities, e.g., to construct defect patterns from multiple similar defect descriptions).

- *Analyze Cause*: Statistics, information, and experiences about the existence of quality defects in the software systems are fed back into the early phases of the software development process to prevent or at least reduce their reoccurrence. Preventive measures include, for example, software requirement inspections or goal-oriented training of employees. Furthermore, information about dependencies between qualities, quality defects, and refactorings are fed back into the quality model development process in order to continuously improve the techniques for quality model development.

Decision Support in Handling

In order to support decisions about what to refactor in a software system, we developed several methods and techniques. The following questions act as the guiding theme for the development and enactment of decision making (i.e., the "plan removal" or "refactor code" phase) as well as understanding (i.e., the "analyze defect" or "document change" phase) in refactoring phases:

- *Decision problem 1*: Which quality defects should be refactored and which might stay in the system?
- *Decision problem 2*: In which sequence should one refactor multiple quality defects in order to minimize effort?
- *Comprehension problem 1*: How does one understand and detect the quality defect in the concrete situation?
- *Comprehension problem 2*: How does one understand the refactoring in the concrete situation and its effect on the software system?
- *Decision problem 3*: Which refactoring should one use if multiple ones are available?
- *Comprehension problem 3*: Which information should one record after refactoring or marking for later evolution, maintenance, or reengineering activities?
- *Decision problem 4*: Did the understanding of the problem or the refactoring result in valuable experience that should be recorded to support later activities (possibly by others)?
- *Comprehension problem 5*: How should one record the experience?

Decision Support in Software Refactoring

Our approach encompasses several methods for supporting the decision of where, when, and in what sequence to refactor a software system as depicted in Figure 4. Beginning from the left upper corner and going counterclockwise, knowledge about quality defects from defect discovery processes is used to retrieve experiences associated with similar defects from previous refactorings. These experiences are used to handle quality defects in the defect removal phase. Additionally, suitable experiences are augmented by so-called micro-didactical arrangements (MDA) (Ras et al., 2005), which initiate learning processes and aim at improving the understandability, applicability, and adaptability of the experience in the specific context.

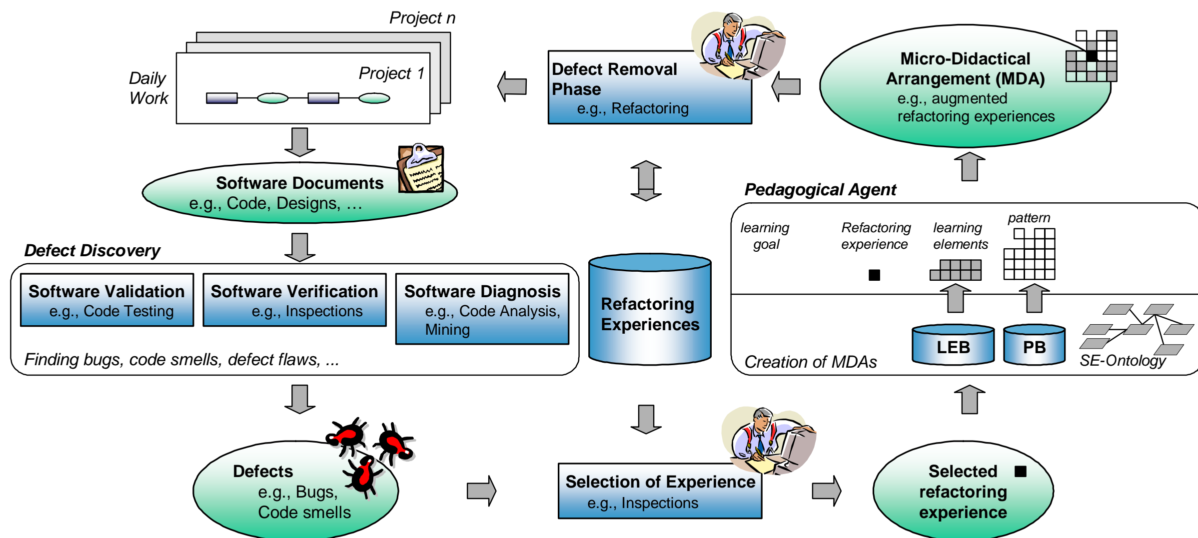

Figure 4. Experience-based semi-automatic reuse of refactoring experiences

As shown in Figure 5, we define six phases, based on the Quality Improvement Paradigm (QIP) (Basili et al., 1994a), for the continuous handling of quality defects. In contrast to the Quality Defect Handling process as depicted in Figure 3, these phases are not concerned with quality defects in a specific product, but with the learning process about the quality defects themselves and their effect on the software qualities. Figure 3 represents the realizations of phase 2 ("Discover defect"), phase 3 ("Plan removal"), and phase 4 (the "Quality Defect Handling" block).

In Figure 5, we first start with the definition of the quality model consisting of qualities that should be monitored and improved. For example, this may result in different goals (i.e., quality aspects), as reusability demands more flexibility or "openness", while maintainability requires more simplicity. Phase two is concerned with the measurement and preprocessing of the source code to build a basis for quality defect discovery (i.e., "Discover Defects"). Results from the discovery process (i.e., quality defects) are represented and prioritized to plan the refactoring in phase three (i.e., "Plan Removal"). Here, the responsible person has to decide which refactorings have to be executed (i.e., "Analyze Defect") in what configuration and sequence, in order to minimize work (e.g., change conflicts) and maximize the effect on a specific quality. In phase four, the refactoring itself is (or is not) applied to the software system (i.e., "Refactor Code" or "Mark Code") by the developer, which results in an improved product. Phase five compares the improved product with the original product to detect changes and their impact on the remaining system (i.e., "Analyze Cause"). Finally, in the sixth phase, we document the experiences and data about the refactoring activity, changes to the software system, and other effects in order to learn from our work and continuously improve the model of relationships between quality, refactorings, and quality defects.

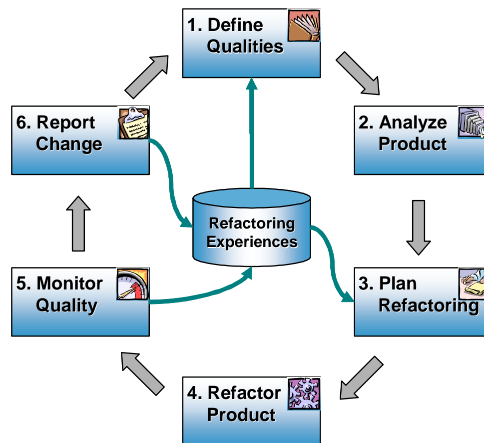

Figure 5. Quality-driven refactoring

As indicated previously, the KDD sub-processes are grouped in phase two. We select source code from a specific build, preprocess the code and store the results in the code warehouse, analyze the data to discover quality defects, discover deviations from average behavior, cluster code blocks with severe or multiple quality defects, and represent discovered and prioritized quality defects to the user.

An Example of DS for QDD

For example, we may detect a method in an object-oriented software system that has a length of 300 LOC. As described in (Fowler, 1999), this is a code smell called Long Method. A Long Method is a problem especially in maintenance phases, as the responsible maintainer will have a hard time understanding the function of this method.

One suitable refactoring for the mentioned code smell might be the refactoring simply called Extract Method: the source code of the Long Method is reviewed to detect blocks that can be encapsulated into new (sub-)methods. Experiences with the Extract Method refactoring are used to support the decision on where, when, how, and if the refactoring has to been implemented. For example, the developer might remark that every block of code that has a common meaning, and could be commented respectively, could also be extracted into several smaller methods. Furthermore, the developer might note that the extraction of (sub-) methods, from methods implementing complex algorithms, can affect the performance requirements of the software system and therefore might not be applicable.

Additionally, the generation of new methods might create another smell called “Large Class” (i.e., the presence of too many methods in a class), which might complicate the case even further. Finally, the new experiences are annotated by the developer and stored in the refactoring experience base.

While this example only touches a simple quality defect and refactoring, more complex refactorings influence inheritance relations or introduce design patterns (Fowler, 1999).

An Annotation Language to Support Quality Defect Handling

This section describes, defines, and explains a language that will be used to annotate code fragments that are either contaminated by quality defects or checked by a software engineer and cleared of quality defects. As described in the background section, several annotation languages for the annotation of source code already exist that are not adequate. This new language is used to keep decisions about quality defects persistent and over time builds a "medical history" of the source code fragment (e.g., a class).

Goals and Characteristics of Annotation Languages

All annotation languages represent a basis for describing additional information about the software system directly at the code level. *Target groups* (or users) for the documentation / annotation language are:

- *Developers,* who want to use the source code and acquire information via the API descriptions (e.g., for software libraries).
- *Testers,* who want to develop test cases and need information about the pre- and post-conditions as well as the functionality to be tested.
- *Maintainers,* who want to evolve the system and need information about existing quality defects, rationales for their persistence (e.g., refactoring would cause loss of performance), or past refactorings (e.g., to update the software documentation such as design documents).

In our case, an annotation language that is targeted at supporting the handling of quality defects should encompass several key aspects. The requirements for such an annotation language should cover uses such as:

- *Annotate change* for later understanding by the same and other readers (e.g., maintainers)
- *Mark fragment* that a quality defect is detected but can or must stay in the system.
- *Note membership* in a larger quality defect or refactoring activity that encompassed multiple code fragments for later impact analyses.
- *Annotate quality aspects* for later reuse, etc.
- *Tag additional information* in the code fragment freely or based on a classification (e.g., "problematic class", "quicksort algorithm", "part of subsystem X") to support later reuse or maintenance/reengineering activities (similar to social software or Web 2.0 approaches)

We identified the following information blocks of an annotation language that should be recorded with an annotation language and that are based on the six knowledge types from Knowledge Management (Mason, 2005):

- *Know-what*: Record the currently present *quality defects* that were found manually or automatically.
- *Know-how*: Record the *transformation history* (similar to the medical history of a human patient).

- *Know-why*: Record the *rationales* why a refactoring was applied or why a quality defect is still present in order to prevent recurrent defect analysis or refactoring attempts.
- *Know-where*: Record the *location* in the annotated code as well as associated code fragments that where changed as well.
- *Know-who*: Record the tool or *person* (i.e., developer or maintainer) who applied the refactoring.
- *Know-when*: Record the time or *version* when the quality defect was found or the refactoring was applied. This could also be used to define a trigger when a refactoring has to be applied (e.g., if several other (larger) refactorings or design decision have to be made).
- *Context*: Record the frame of reference or context in which the quality defect was discovered. This includes especially the quality model used to decide which quality defect has a higher priority over other quality defects.

The following requirements for tags and other constructs in such an annotation language to support refactoring and maintenance activities are:

- *Unambiguous*: The names of tags, quality defects, refactorings, or other reoccurring terms should be unique and used consistently throughout the system.
- *Machine-readable*: The syntax of tags should be formal, exact, and consistent to avoid confusion and enable the interpretation and usage by supporting systems (e.g., defect discovery tools).
- *Local completeness*: The power of the syntax should be large enough to cover all existing cases. Full comprehensiveness is probably not possible except by allowing informal free text attributes.
- *Flexibility*: The syntax should not limit the extension by new tags or tag attributes.
- *Independence*: Tags should describe information that is mutually exclusive, and the occurrence of two or more tags should be independent from one another.

Beside the additional documentation of the software system, the annotation language will increase the semantic coupling between code fragments and reduce the presence of quality defects such as “shotgun surgery”.

RAL: The Refactoring Annotation Language

The refactoring annotation language (RAL) is used to record the currently existing quality characteristics, symptoms, defects, and refactoring of a code fragment regarding a specific quality model. Furthermore, it is used to store the rationales and treatment history (e.g., sequence of refactorings).

In the following tables, the core set of tags from RAL are described based on the JavaDoc syntax and using existing JavaDoc and supportive tags, that are used in the description and will be described after the core tags. Information blocks starting with a double cross “#” indicate an ID or standardized term from an external, controlled vocabulary or taxonomy.

A symptom tag as defined in Table 7 describes a metric or characteristic of the code fragment and is used as an indicator for the statistical or rule-based identification of quality defects. The tag acts as an annotation of a specific symptom from a controlled vocabulary in order to have a unique identifier and a reference for further information about the symptom. The since tag from JavaDoc is used to identify the version based on which the quality symptom was first calculated.

Table 7 The @symptom Tag

Tag Syntax	@symptom <#Symptom-ID> <@value value> <@since #version>
Example	@symptom "LOC" @value "732" @since 1.2

The quality defect as defined in Table 8 represents a code smell, antipattern, etc. present in this code fragment. It is used to annotate a specific quality defect from a controlled vocabulary in order to have a unique identifier and reference for more information about a specific quality defect type and potential treatments. The since tag from JavaDoc is used to identify the version where the quality defect was first noticed.

Table 8 The @defect Tag

Tag Syntax	@defect <#QD-ID> <@since #version> <@status #Status> <@rationale text>
Example	@defect "Long Method" @since 1.2 @status "untreated"

A refactoring tag as defined in Table 9 is a description of a single refactoring that was applied for removing one or more quality defects. Optionally, a project-internal URI to other code fragments directly affected by the refactoring (e.g., if two classes interchange a method during the same refactoring) can be stated.

Table 9 The @refactoring Tag

Tag Syntax	@refactoring <#Refactoring-ID> <@rationale text> <@status #Status> <@link fragment> <@author name>
Example	@refactoring "Extract Method" "Applied as quality model rates maintainability higher than performance" @status "treated" @link "ExtractedMethod" @author "John Doe"

The quality model tag as defined in Table 10 is used as a reference to the quality model that defines which quality characteristics are important, what priority or decision model lies beneath, and which quality defects are relevant to a specific part of the software system. Optionally, it refers to a URI of a file containing the specific (machine-readable) quality model.

Table 10 The @quality-model Tag

Tag Syntax:	@quality-model Name <@see file>
Example	@quality-model "QM-Dep1-Project2" @see <A href=http://www.sw-quality.de/QM-Dep1-Project2.pdf>

The supportive tags used in the above tag descriptions are given in Table 11.

Table 11 Support Tags

Tag	Description
@status	"@status #status" indicates the current status of the superior tag or source code using the vocabulary "discovered", "inWork", "treated", or (deliberately) "untreated".
@rationale	"@rationale text" declares a rationale about the existence or status of the superior tag or source code.

Depending on the processor that would render a quality documentation from these tags, some tags might be used only once and inherited by lower levels. For example, the quality model tag needs only be stated once (e.g., for the whole project) or twice (e.g., for the client and server part) in a software system.

RAL is primarily used to annotate source code. Therefore, in order to annotate documents of higher abstraction, like UML-based design documents (e.g., platform-independent models in MDA) using the XMI Format or formal requirement documents, similar languages (probably based on other languages such as JavaDoc) need to be defined.

Handling Quality Defects Using RAL

Software annotation languages like JavaDoc or Doxygen and extensions like RAL can now be used to document the functionality, structure, quality, and treatment history of the software system at the code level. The formal basis of the language enables tools to read and write this information automatically to generate special documents or trigger specific actions.

The core tags `@symptom`, `@defect`, and `@refactoring` build on top of each other and might be recorded by several different tools. This enables the intertwined cooperation of different tools, each with a specific focus, such as to calculate metrics or to discover quality defects. For example, one tool might measure the source code and its versions to extract numerical and historical information and write it into `@symptom` tags (e.g., Lines of Code). Another tool might analyze this information to infer quality defects (e.g., "Long Method") that are recorded in `@defect` tags. Finally, a tool might offer refactorings to a developer or a maintainer during his work and note applied refactorings or rationales in explicit `@refactoring` tags.

Developers and maintainers of a software system are supported in the handling of quality defects in the following activities:

- Repetitive refactoring of a specific kind of quality defect (e.g., "Large Method"), as they do not have to switch between different defects or refactoring concepts.
- Reuse of knowledge about the refactoring of specific quality defects to judge new quality defects.
- Recapitulation of the change history of the code fragment to update software documentation such as design documents.
- Retrieval of information about persons who developed or refactored this part of the system and should know about its purpose and functionality.

Product or quality managers of the software system might use the information to:

- Evaluate the quality based on information extracted via the tags about the amount or distribution of quality defects.
- Analyze specific dates or groups of persons that might have introduced specific kinds of quality defects and might need further training.

Summary & Outlook

Agile software development methods were invented to minimize the risk of developing low-quality software systems with rigid process-based methods. They impose as little overhead as possible in order to develop software as fast as possible and with continuous feedback from the customers. To assure quality, agile software development organizations use activities such as refactoring between development iterations. *Refactoring,* or the restructuring of a software system without changing its behavior, is necessary to remove *quality defects* (i.e., bad smells in code, architecture smells, anti-patterns, design flaws, software anomalies, etc.) that are introduced by quick and often unsystematic development. However, the effort for the manual discovery of these quality defects results in either incomplete or costly refactoring phases. Furthermore, software quality assurance methods seem to ignore their recurring application.

In this chapter, we described a process for the recurring and sustainable discovery, handling, and treatment of quality defects in software systems. We described the complexity of the discovery and handling of quality defects in object-oriented source code to support the software refactoring process. Based on the formal definition of quality defects, we gave examples of how to support the recurring and sustainable handling of quality defects. The annotation language presented is used to store information about quality defects found in source code and represents the defect and treatment history of a part of a software system. The process and annotation language can not only be used to support quality defect discovery processes, but also has the potential to be applied in testing and inspection processes.

Recapitulating, we specified an annotation language that can be used in agile software maintenance and refactoring to record information about quality defects, refactorings, and rationales about them. Similar annotation languages such as JavaDoc or doxygen as well as third-party extensions are not able to encode this information in a machine-readable and unambiguous format.

The proposed framework including the handling process promises systematic and semi-automatic support of refactoring activities for developers, maintainers, and quality managers. The approach for recording quality defects and code transformations in order to monitor refactoring activities will make maintenance activities simpler and increase overall software quality. Likewise, the user monitors daily builds of the software to detect code smells, identical quality defects or groups thereof, and initiates repetitive refactoring activities, minimizing effort caused by task switches.

Requirements for Quality Defect Handling in Agile SE

When building systems and languages for quality defect handling in agile software development, several requirements should be kept in mind.

The annotation language in the form of a code annotation language like JavaDoc or in the form of an external documentation such as a Defect Tracking system or a Wiki should be integrated into the programming language used and into the development environment. If it is not integrated, the information might easily be lost due to the high workload and time constraints in agile development. Especially in an agile environment, the developers, testers, and maintainers should be burdened with as little additional effort as possible.

Therefore, the more formal the descriptions of an annotation language are and the more information can be extracted from the code and development environment (e.g., from the refactoring techniques), the less information is required from the developers.

Outlook

The trend in research is to increase automation of the mentioned processes in order to support the developers with automated refactoring or defect discovery systems.

We expect to further assist software engineers and managers in their work and in decision making. One current research task is the development of taglets and doclets to generate specific evolution documents. Furthermore, we are working on the analysis and synthesis of discovery techniques with statistical and analytical methods based on textual, structural, numerical, and historical information.

Although we can record and use this information in several applications, we currently do not know if the amount of information might overwhelm or annoy the developer and maintainer. If dozens of quality defects are found and additional refactorings are recorded, this might be confusing and should be hidden (e.g., in an editor of the IDE) from the developer. Very old information (e.g., from previous releases of the software) might even be removed and stored in an external document or database.

References

Allen, E. (2002). Bug patterns in Java. Berkeley: Apress, USA, New York, NY.

Atlassian. (2005). JIRA Website. Retrieved 6. Oct., 2005, from http://www.atlassian.com/software/jira/

Aurum, A., Petersson, H., & Wohlin, C. (2002). State-of-the-art: software inspections after 25 years. Software Testing, Verification and Reliability, 12(3), 133-154.

Basili, V. R., Caldiera, G., & Rombach, D. (1994a). The Goal Question Metric Approach. In Encyclopedia of Software Engineering (1st Edition ed., pp. 528-532). New York: John Wiley & Son.

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994b). Experience Factory. In J. J. Marciniak (Ed.), Encyclopedia of Software Engineering (Vol. 1, pp. 469-476). New York: John Wiley & Sons.

Beck, K. (1999). eXtreme Programming eXplained: Embrace Change. Reading: Addison-Wesley.

Beck, K., & Fowler, M. (1999). Bad Smells in Code. In G. Booch, I. Jacobson & J. Rumbaugh (Eds.), Refactoring: Improving the Design of Existing Code (1st ed., pp. 75-88): Addison-Wesley Object Technology Series.

Brown, W. J., Malveau, R. C., McCormick, H. W., & Mowbray, T. J. (1998). AntiPatterns: refactoring software, architectures, and projects in crisis. New York: John Wiley & Sons, Inc.

Brykczynski, B. (1999). A survey of software inspection checklists. Software Engineering Notes, 24(1), 82-89.

Chillarege, R. (1996). Orthogonal Defect Classification. In M. R. Lyu (Ed.), Handbook of software reliability engineering (pp. xxv, 850 p.). New York: IEEE Computer Society Press.

Ciolkowski, M., Laitenberger, O., Rombach, D., Shull, F., & Perry, D. (Year). Software inspections, reviews and walkthroughs. Paper presented at the 24th International Conference on Software Engineering (ICSE 2002), New York, NY, USA, Soc.

Dromey, R. G. (1996). Cornering the Chimera. IEEE Software, 13(1), 33-43.

Dromey, R. G. (2003). Software Quality—Prevention versus Cure? Software Quality Journal, 11(3), 197-210.

ePyDoc. (2005). Epydoc website. Retrieved 5.10.2005, 2005, from http://epydoc.sourceforge.net/

Fenton, N. E., & Neil, M. (1999). Software metrics: successes, failures and new directions. Journal of Systems and Software, 47(2-3), 149-157.

Force10. (2005). Software Support Management System (SSM) Website. Retrieved 6. Oct., 2005, from http://www.f10software.com/

Fowler, M. (1999). Refactoring: Improving the Design of Existing Code (1st ed.): Addison-Wesley.

Freimut, B. (2001). Developing and Using Defect Classification Schemes (Technical Report No. IESE-Report No. 072.01/E). Kaiserslautern: Fraunhofer IESE.

Fukui, S. (Year). Introduction of the software configuration management team and defect tracking system for global distributed development. Paper presented at the 7th European Conference on Software Quality (ECSQ 2002), Helsinki, Finland, 9-13 June 2002.

Gamma, E., Richard, H., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software (3rd printing ed. Vol. 5): Addison-Wesley.

Hallum, A. M. (2002). Documenting Patterns. Unpublished Master Thesis, Norges Teknisk-Naturvitenskapelige Universitet.

IEEE-1044. (1995). IEEE guide to classification for software anomalies, IEEE Std 1044.1-1995.

Johnson, J. N., & Dubois, P. F. (2003). Issue tracking. Computing in Science & Engineering, USA * vol 5 (Nov. Dec. 2003), no 6, p 71 7.

JSR-260. (2005). Javadoc Tag Technology Update (JSR-260). Retrieved 6th October, 2005, from http://www.jcp.org/en/jsr/detail?id=260

Kerievsky, J. (2005). Refactoring to patterns. Boston: Addison-Wesley.

Koru, A. G., & Tian, J. (2004). Defect handling in medium and large open source projects. IEEE Software, 21(4), 54-61.

Kramer, D. (Year). API documentation from source code comments: a case study of Javadoc. Paper presented at the 17th International Conference on Computer Documentation (SIGDOC 99), New Orleans, LA, USA, 12-14 Sept. 1999.

Kramer, R. (1998). iContract - The Java(tm) Design by Contract(tm) Tool. In Technology of Object-Oriented Languages and Systems, TOOLS 26 (pp. 295-307). Santa Barbara, CA, USA: IEEE Computer Society.

Lauesen, S., & Younessi, H. (1998). Is Software Quality visible in the code? IEEE Software, 15(4), 69-73.

Liggesmeyer, P. (2003). Testing safety-critical software in theory and practice: a summary. IT Information Technology, 45(1), 39-45.

Loper, E. (2004). Epydoc: API Documentation Extraction in Python, from http://epydoc.sourceforge.net/pycon-epydoc.pdf

Mantis. (2005). Mantis Website. Retrieved 6. Oct., 2005, from http://www.mantisbt.org/

Marinescu, R. (Year). Detection strategies: metrics-based rules for detecting design flaws. Paper presented at the 20th International Conference on Software Maintenance, Chicago, IL, USA, 11-14 Sept. 2004.

Mason, J. (2005). From e-learning to e-knowledge. In M. Rao (Ed.), Knowledge Management Tools and Techniques (Paperback ed., pp. 320-328). London: Elsevier.

Mens, T., Demeyer, S., Du Bois, B., Stenten, H., & Van Gorp, P. (2003). Refactoring: Current Research and Future Trends. Electronic Notes in Theoretical Computer Science, 82(3), 17.

Mens, T., & Tourwe, T. (2004). A survey of software refactoring. IEEE Transactions on Software Engineering, 30(2), 126-139.

MetaQuest. (2005). Census Website. Retrieved 6th October, 2005, from http://www.metaquest.com/Solutions/BugTracking/BugTracking.html

Pepper, D., Moreau, O., & Hennion, G. (Year). Inline automated defect classification: a novel approach to defect management. Paper presented at the IEEE/SEMI Advanced Semiconductor Manufacturing Conference and Workshop, Munich, Germany, 11-12 April 2005.

Rapu, D., Ducasse, S., Girba, T., & Marinescu, R. (Year). Using history information to improve design flaws detection. Paper presented at the Eighth European Conference on Software Maintenance and Reengineering, Tampere, Finland.

Ras, E., Avram, G., Waterson, P., & Weibelzahl, S. (2005). Using Weblogs for Knowledge Sharing and Learning in Information Spaces. Journal of Universal Computer Science, 11(3), 394-409.

Rech, J. (Year). Towards Knowledge Discovery in Software Repositories to Support Refactoring. Paper presented at the Workshop on Knowledge Oriented Maintenance (KOM) at SEKE 2004, Banff, Canada.

Rech, J., & Ras, E. (Year). Experience-Based Refactoring for Goal-Oriented Software Quality Improvement. Paper presented at the First International Workshop on Software Quality (SOQUA 2004), Erfurt, Germany.

Remillard, J. (2005). Source code review systems. IEEE Software, 22(1), 74-77.

Riel, A. J. (1996). Object-oriented Design Heuristics. Reading, Mass.: Addison-Wesley Pub. Co.

Roock, S., & Havenstein, A. (Year). Refactoring Tags for automatic refactoring of framework dependent applications. Paper presented at the Extreme Programming Conference XP 2002, Villasimius, Cagliari, Italy.

Roock, S., & Lippert, M. (2004). Refactorings in großen Softwareprojekten: Komplexe Restrukturierungen erfolgreich durchführen (in German). Heidelberg: dpunkt Verlag.

Roock, S., & Lippert, M. (2005). Refactoring in Large Software Projects (Paperback ed.): John Wiley & Sons.

Sametinger, J., & Riebisch, M. (Year). Evolution support by homogeneously documenting patterns, aspects and traces. Paper presented at the Sixth European Conference on Software Maintenance and Reengineering, Budapest, Hungary, 11 13 March 2002.

Serrano, N., & Ciordia, I. (2005). Bugzilla, ITracker, and other bug trackers. IEEE Software, 22(2), 11-13.

Simonis, V., & Weiss, R. (Year). PROGDOC - a new program documentation system. Paper presented at the 5th International Andrei Ershov Memorial Conference (PSI 2003) Perspectives of System Informatics, Novosibirsk, Russia, 9-12 July 2003.

Tigris. (2005). Scarab Website. Retrieved 6. Oct., 2005, from http://scarab.tigris.org/

Torchiano, M. (Year). Documenting pattern use in Java programs. Paper presented at the Proceedings of the International Conference on Software Maintenance (ICSM), Montreal, Que., Canada, 3-6 Oct. 2002.

Tourwe, T., & Mens, T. (2003). Identifying refactoring opportunities using logic meta programming. IEEE Computer, Reengneering Forum; Univ. Sannio. - In Proceedings Seventh European Conference on Software Maintenance and Reengineering. - Los Alamitos, CA, USA, USA IEEE Comput. Soc, 2003, xi+2420 2091-2100, 2031 Refs.

TRAC. (2005). TRAC Website. Retrieved 6th October, 2005, from http://projects.edgewall.com/trac/

Tullmann, P. (2002). Pat's Taglet Collection. Retrieved 2005, 6th October, from http://www.tullmann.org/pat/taglets/

Tuppas. (2005). Tuppas Website. Retrieved 6. Oct., 2005, from http://www.tuppas.com/Defects.htm

van Heesch, D. (2005). Doxygen - a documentation system., from http://www.doxygen.org/

Verhoef, C. (Year). How to implement the future? Paper presented at the Proceedings of the 26th EUROMICRO Conference (EUROMICRO'2000), Maastricht, The Netherlands, September 05 - 07, 2000.

Wake, W. C. (2003). Refactoring Workbook (1st ed.): Pearson Education Inc.

Whitmire, S. A. (1997). Object-oriented Design Measurement. New York, NY, USA: John Wiley & Sons.