SPECIFIC TARGETED RESEARCH PROJECT
INFORMATION SOCIETY TECHNOLOGIES

FP6-IST-2004-033606

VIsualize all moDel drivEn programming
VIDE

WP 4	Deliverable Number D.4.1 Quality Defects in Model-driven Software Development

Project name:	Visualize all model driven programming
Start date of the project:	01 July 2006
Duration of the project:	30 months
Project coordinator:	Polish - Japanese Institute of Information Technology
Work package Leader:	Fraunhofer IESE
Due date of deliverable:	30. June 2007
Actual submission date	09. August 2007
Status	**final**
Document type:	Report
Document acronym:	D4.1

Editor(s)	Jörg Rech, Axel Spriestersbach
Reviewer(s)	Jörg Rech, Axel Spriestersbach, Andreas Jedlitschka, Sonnhild Namingha, Andreas Roth
Accepting	Kazimierz Subieta
Location	http://www.vide-ist.eu
Version	1.0
Dissemination level	**PU (Public)**

Abstract

To support the modeler of a PIM during his work, information about possible threats to the quality (in respect to ISO 9126) of the PIM should be identified as early as possible. In this work package quality defects such as architectural smells, anti-patterns, or design flaws are investigated that might occur on the PIM level, and especially in the behavior model via the VIDE action language. Furthermore, new quality defects are explored that might emerge in the data-intense business domain or in the general context of action languages. The knowledge explored in this work package will be used to develop a module of VIDE that discovers quality defects in the internal PIM representation and annotates its textual and visual representation.

This deliverable is split into two parts. This report provides an in-depth survey of the state-of-the-art in quality defects that are a potential threat to the quality of models in MDSD.

The second report will focus on quality defect discovery techniques and include the technical specification of techniques for discovering quality defects as well as methods for handling them.

The VIDE consortium:

Polish-Japanese Institute of Information Technology (PJIIT)	**Coordinator**	**Poland**
Rodan Systems S.A.	Partner	Poland
Institute for Information Systems at the German Research Center for Artificial Intelligence	Partner	Germany
Fraunhofer-Gesellschaft e.V.	Partner	Germany
Bournemouth University	Partner	United Kingdom
SOFTEAM	Partner	France
TNM Software GmbH	Partner	Germany
SAP AG	Partner	Germany
ALTEC	Partner	Greece

Table of Contents

List of Figures

List of Tables

1 Introduction and Overview

Model-driven software development (MDSD) drastically alters the software development process, which is characterized by a high degree of innovation and productivity. MDSD focuses on the idea of constructing software systems not by programming in a specific programming language, but by designing models that are translated into executable software systems by generators. These characteristics enable designers to deliver product releases within much shorter periods of time and develop more different platforms compared to the traditional methods. In theory, this process makes it unnecessary to worry about an executable system's quality, as it is "optimized" by the generators.

However, proponents of MDA must provide convincing answers to questions such as "What is the quality of the models and software produced?" The designed models are also a work product that requires a minimal set of quality aspects (e.g., the maintainability of models over a longer life-time). Quality assurance techniques such as testing, inspections, software analysis, or software measurement are well researched for programming languages, but their application in the domain of software models and model-driven software development is still in an embryonic phase.

The goals of quality assurance for model-driven software development are diverse and include the improvement of quality aspects such as maintainability, reusability, security, or performance. Quality assurance for model-driven software development will play an important role for the future wide-spread usage of model-driven architectures in general, as well as in specific application domains.

The main concern of software quality assurance (SQA) is the efficient and effective development of large, reliable, and high-quality software systems. While verification and validation efforts in industry typically focus on functional aspects, using techniques such as testing or inspection, other quality aspects are often neglected. However, the non-functional quality of a software product is crucial for its evolution and maintenance by the same or another software organization. Other techniques such as software product analysis and measurement are either used to measure software systems and interpret their quality based on a previously defined quality model or to predict project characteristics based on experiences from past measurements. From the deficits found by interpreting the quality characteristics (e.g., software metrics), further actions are derived on an abstract level to improve software quality.

Another approach in SQA is the diagnosis of explicitly defined defects such as anti-patterns, design flaws, or code smells, which represent system-independent defects with a negative effect on a quality aspect such as maintainability. Individual refactorings are used to remove these defects and improve the defective parts without changing its functionality.

Today, a vast number of these defects are known and documented in various communities under various names. Typically, they are collected and described by practitioners and consultants and represent condensed experiences from multiple projects they were involved in. In this report, the term *quality defect* is used as an umbrella term for the concepts antipattern, smell, flaw, pitfall, bug pattern, defect pattern, negative pattern, (bad) heuristic, (bad) characteristic, anti-idiom, (design) problem, (design) defect, refactoring candidate, puzzlers, traps, anomalies, and many more (typically with an additional focus on a quality aspect, development phase, or abstraction level – e.g., a performance antipattern, test smell, or architectural anomaly) that have a negative effect on a quality aspect (e.g., maintainability or reusability). Problems concerning the compilability of the model (e.g., missing attributes) or regarding the conformance to a standard (e.g., capitalize class name) are not in the focus of quality defects.

In spite of the large number of quality defect collections available today, not many techniques, methods, or tools are available for their manual, semi-manual, and automatic diagnosis (i.e., their diagnosis or prognosis). On the one hand, this might be the result of them being described in different formalization grades, the formal and complete representations of the defective objects (e.g., a software project plan) not being adequate, and not all of them are diagnosable at all. On the other hand, most practitioners and researchers are neither aware of the various concepts quality defects are known under nor are they informed about all defects under one concept.

To support the modeler of a platform-independent model (PIM) during his work, information about possible threats to the quality (in respect to ISO 9126) of the PIM should be identified as early as possible. In this work package, quality defects are investigated that might occur on the PIM level, and especially in the behavior model via the VIDE action language. Furthermore, new quality defects are explored that might emerge in the data-intense business domain or in the general context of action languages. The knowledge explored in this work package will be used to develop a module of VIDE that discovers quality defects from the internal PIM representation and annotates its textual and visual representation.

In contrast to the insular and inconsistent collections in other publications this report presents the results of a systematic literature review to create a comprehensive and uniform collection of these quality defects and start a quality defect body of knowledge. We have selected over 560 black and grey publications published in scholarly literature. This review includes a summary of quality defects and their definitions. The sister report D4.2 will include a summary of quality defect diagnosis techniques, their characteristics, benefits, and shortcomings.

1.1 The objectives of WP4

The results presented in this report are based upon a systematic literature review that was targeted to be complete, concise, and consistent. It is the result of tasks 4.1, 4.2, and 4.3 as defined in the project and listed in the following:

- **Task 4.1 Researching and summarizing existing quality defects** (Task leader IESE): A detailed analysis of the state-of-the-art in quality defect discovery (which is currently largely done on source code) will be performed to develop a strong foundation for the later work. Another goal is to elicit a summary of existing quality defects that might appear in higher levels of the software development process and especially on PIM and PSM.
- **Task 4.2 Modeling the information- and defect model for MDA** (Task leader IESE): In order to identify quality defects in a PIM, a formal model to describe the morphology (i.e., the inner structure and characteristics) of quality defects will be defined. Based upon this formal defect model and the definition of the VIDE language (i.e., the representation of a PIM) the information model that describes the available information that a tool can use to identify potential quality defects will be synthesized.
- **Task 4.3 Modeling domain-specific parts of the models** (Task leader SAP): To identify and formalize quality defects specific to a particular domain of business applications, the domain-specific variabilities of the domain with respect to quality will be analyzed and compared against the defect model. This will result in an extension (i.e., variant) of the core defect model for a specific domain summarizing and characterizing quality defects of this domain.
- **Task 4.4 Development of techniques for PIM-specific quality defects** (Task leader IESE): Based upon the required information that characterizes quality defects (i.e, the defect model) and the available information from the PIM (i.e., the information model),

techniques for automatically discovering symptoms that indicate specific quality defects will be defined.

- **Task 4.5 The quality defect discovery module** (Task leader IESE): Based upon the techniques of quality defect discovery, the visual editor, and the general process, the tool for quality defect discovery that will support the modeler of a PIM will be designed. The design of the discovery techniques will be based either upon the standard languages (e.g., OCL) used or on free parsing and reasoning technologies (e.g., ANTLR).

In work package D.1 ("Standards, Technological and Research-Base for the VIDE Project, Project Evaluation Criteria and User Requirements Definition"), the consortium has investigated the typical user groups for the VIDE environment. All these user groups have potentially different requirements on the visualization of the model (esp. the PIM) and, in our context, the visualization of quality defects regarding structural and behavioral aspects of the model.

The identified user roles "analysts/designers", "analysts/VIDE programmers", and "architects" are all strongly integrated into the PIM modeling process. However, the two other roles "domain users" and "business analysts" (resp. "requirement analysts") are not necessarily required to develop the software model on the PIM level. They are more involved in the development of the CIM level and might support others in the development of the PIM (e.g., as contacts for the information encoded in the CIM).

The core roles involved in the development of the PIM are the analysts/designers, analysts/VIDE programmers, and architects. They, as well as their variants (e.g., GUI designer, DB tester, etc.), are responsible for the creation, modification, and quality assurance of the PIM. Based on the description in D.1, they have the following foci that should be supported by the visualization of quality defects:

- **Analysts/designers** are responsible for the conceptual platform-independent model that is based on the computational-independent model produced by the business analyst. This role uses the VIDE language and tools to define the first level of behavior, but leaves the details to the VIDE developer. Therefore, it requires information on the quality of the model regarding the big picture (i.e., *architecture*) as well as *structural aspects*. Additionally, as this role is also responsible for deciding if predefined components may be reused it has to have information regarding the components' *interfaces* as well as regarding the reusability, adaptability, or composability of the components.
- **Analysts/VIDE programmers** are responsible to complete the models regarding *behavioral aspects* in such a way that will allow model simulation (i.e., for testing) and transformation of the models into code. This role is only marginally concerned with structural aspects developed by the analysts/designers.
- **Architects** are responsible for building the transformations of the PIM described using VIDE into platform specific models and code. Additionally, the architect is an expert in the target platform (e.g., Java VM, Tomcat/JSP, Struts, etc.) but also has a good understanding of UML and VIDE in order to be able to define the transformation. This role is required to work with the complete PIM and PSM.

In summary, this report provides an extensive overview of existing quality defects affecting quality aspects of software products, processes, projects, and organizations as well as techniques for their diagnosis. Section 2 describes the background of software quality assurance and quality defects, while section 3 contains the design of the systematic literature survey. In particular, section 4 contains the description of quality defects and related concepts that were described in the literature in order to describe the objects of diagnosis. Section 5 presents information on the data-oriented business domain that is targeted in the context of the VIDE project. The selected quality defects that are to be used in WP9 are described in section 6.

Finally, the conclusion in section 7 summarizes this report and gives an outlook on current research and trends.

2 Background

The software industry has a reputation for producing expensive, low-quality software as software systems have reached a level of complexity that puts them beyond our ability to evolve and maintain them easily. This increases the need for software organizations to develop or rework existing systems with high quality.

To improve the quality of their software products, organizations often use quality assurance activities such as refactoring of the source code to tackle defects that reduce internal or external quality aspects of the software. During the last years many practitioners recorded their experience with these kinds of defects in form of patterns and antipatterns (i.e., recurring solutions or problems). However, only few of these collections are known to the research community. Most of the developed approaches concerning such defects do only take code smells, design flaws, and antipatterns into consideration. A comprehensive collection of the quality defects will hopefully foster the research in this area.

Today, several types of *quality defects* (i.e., smells, anti-patterns, flaws, bug patterns, pitfalls, etc.) can be diagnosed on the code level but also exist as threats to the quality of earlier abstractions of the software system such as software models. While several approaches were developed in the past to diagnose these quality defects in the source code of software systems, the diagnosis of quality defects in software models (esp. in model abstractions used in MDSD such as PIMs) is underdeveloped. Especially, the richness of information available in software models other than class diagrams has still not been made available for quality defect diagnosis. Furthermore, the dependency of the context of a quality defect has not been analyzed deeply. Several quality defects are location sensitive in such a way that they might emerge during the application of an architectural style or design pattern (e.g., a Large Class in a façade pattern), a contextual convention (e.g., the TypeEmbeddedInName smell in Java's "toString" methods), or other best practices.

2.1 Introduction to SQA

The techniques to diagnose quality defects are based upon research from the fields *software refactoring* (Fowler, 1999; Mäntylä, 2003; Mens & Tourwe, 2004; Simon et al., 2001; Tahvildari et al., 2003; van Emden & Moonen, 2002) to diagnose and remove quality defects, *software inspections* (Aurum et al., 2002; Ciolkowski et al., 2002; Wohlin et al., 2002) to manually detect and analyze ambiguities in analysis or coding phases, *source code analysis* (Fenton & Neil, 1999; Fenton & Ohlsson, 2000) to quantify code characteristics for quality measurement and assurance, and *software testing* (Liggesmeyer, 2003) to detect functional defects after implementation.

While some techniques for the diagnosis of quality defects in source code are already known, the diagnosis of quality defects based on architectural and design information used in model-driven software development (MDSD) and especially platform-independent models (PIMs) from early software development phases are not well understood and open to further investigation. Furthermore, with the rise of MDSD the need for high-quality and maintainable software models will increase.

In VIDE, quality assurance knowledge for platform-independent models will be researched to increase their quality and ease the development and maintenance of these models. This knowledge will be used to enrich the visualization of the models in order to inform the designers and maintainers about potential threats to model quality.

The remainder of this section describes the background of quality assurance for MDSD with a focus on quality defect diagnosis that is needed in the VIDE research project (especially in WP4). This overview was developed in Task 4.1 and summarizes the core concepts of quality defects and quality defect diagnosis.

2.2 Software Quality

Today, the quality of software systems is very important in the development of software systems. While quality factors can be identified for every product, process, project, or person in a software engineering organization the focus in this work package of the VIDE project lies on the software product quality.

The quality of software systems can be subdivided into several smaller aspects that focus on specific characteristics such as maintainability, performance, or usability. These quality aspects have two main addressees.

- The first addressee is the user of the software system who typically emphasizes quality aspects such as usability or performance. Quality aspects mainly concerning the users are also called *external quality aspects*. External quality aspects are typically defined by the customer or through a user survey and codified in non-functional requirements.
- The second addressee is the developing software organization that emphasizes quality aspects such as maintainability or portability of the system expressed as source code. These quality aspects are called *internal quality aspects*. On the model layer similar quality aspects exists that describe that emphasize regarding the architects and analysts.

Other addresses of quality aspects are, for example, system operators (e.g., administrators) that need an easily installable system. But in general every person involved in development, administration, or usage activities of the software system has own specific quality aspects.

Today, many quality aspects of various granularity are defined and used differently in quality models. Several of these quality aspects, that are relevant to this report, are described in the following (excluding the "compliance" sub-characteristics) to give an impression of their focus:

- **Maintainability**: This quality aspect describes how easy or difficult it is to correct, adapt, or perfect the software system. In (IEEE-610, 1990) maintainability is defined as "*the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment.*" In (ISO/IEC-9126-1, 2003) maintainability is defined as "*A set of attributes that bear on the effort needed to make specified modifications*" and sub-divided into the sub-characteristics Stability, Analyzability, Changeability, and Testability. Other sub-characteristics that might be associated with maintainability are comprehensibility or inspectability (resp. reviewability).
- **Reusability**: This quality aspect describes how easy it is to reuse the system in another software system or a variant of the software system. In (IEEE-610, 1990) reusability is defined as "*the degree to which a software module or other work product can be used in more than one computing program or software system.*" In (ISO/IEC-9126-1, 2003) reusability is not defined as a quality characteristic or sub-characteristic.
- **Performance**: This quality aspect describes how fast the software system processes a task and how fast it reacts on (user) input. In (IEEE-610, 1990) it is defined as "the degree to which a system or component accomplishes its designated functions within given constraints". In (ISO/IEC-9126-1, 2003) performance is a part of efficiency that is defined as "a *set of attributes that bear on the relationship between the level of performance of the*

software and the amount of resources used, under stated conditions" and sub-divided into the sub-characteristics Time Behavior and Resource Utilization.

- **Portability**: This quality aspect describes how easy it is to port, migrate, or recompile the software system on a new platform. In (IEEE-610, 1990) portability is defined as "*the ease with which a system or component can be transferred from one hardware or software environment to another*". In (ISO/IEC-9126-1, 2003) portability is defined as "*A set of attributes that bear on the ability of software to be transferred from one environment to another*" and sub-divided into the sub-characteristics Installability, Replaceability, Co-Existence, and Adaptability.

Beside these quality aspects several other aspects of software models are important during their development that have not an effect on the quality of the software (or model):

- **Conformance:** This aspect describes if the model complies with a defined set of specifications such as the well-formedness rules in UML or the Java specifications.
- **Compilability:** This aspect describes if the model might be used by a generator or transformator to compiled it into a PSM or code model (e.g., Java).

If one wants to improve any of these aspects he first has to measure it and then apply techniques that improve the status. Methods like GQM (Basili et al., 1994) give support in the definition of metrics but one has to be careful not only to improve the values of the measured metrics (i.e., address the symptoms).

Dromey suggests that in order to identify what quality aspect you want to improve one has to find the corresponding "tangible properties" for the code (Dromey, 1996). A tangible property is a property of the source code that one can measure using knowledge about the programming language, hardware, and software environment.

Definition 1 ***Dromey's Construction Theorem***

A violation of a tangible quality-carrying property will cause a quality defect in the product. Any quality defect can be traced to a violation of a tangible, quality-carrying property.

2.2.1 Software Quality Models

Several models to describe and systematize software quality have been developed during the last forty years to support the communication, planning, controlling, and assessment of software systems. Typically, the quality aspects as described in section 2.2 are used to create a set of interrelated quality aspects that describe how a "good" or "healthy" software system of a specific type (e.g., embedded driving assistance) should look like.

In order to improve the quality of a software systems first it has to be defined what quality means in the specific context. One quality aspect (e.g., performance) might be of utmost importance to a software system in one context (e.g., life-critical situations) but relatively irrelevant in another (e.g., compiler). Based upon a general quality model a product-specific quality model has to be instantiated.

Several of these general quality models were developed until today as shown in Table 1. The international standard ISO/IEC 9126 (ISO/IEC-9126-3, 2004) represents a general approach that defines a quality model for software products. While there exists some critique about if ISO/IEC 9126 categorization is correct and reliable in evaluating user satisfaction (Ho et al., 2004) it is constantly improved. Currently, the new Standard (ISO/IEC-25000, 2005) is being developed in the SQuaRe project that is targeted to replace ISO-9126.

Table 1. *Quality Aspects used in the different Quality Models (based on (Ortega et al., 2003))*

	Boehm	McCalls	FURPS	ISO 9126	Dromey
Testability	x	x		x	
Correctness		x			
Efficiency	x	x	x	x	x
Understandability	x			x	
Reliability	x	x	x	x	x
Flexibility		x	x		
Functionality			x	x	x
Human Engineering	x				
Integrity/Security		x		x	
Interoperability		x		x	
Process Maturity					x
Maintainability	x	x	x	x	x
Changeability	x				
Portability	x	x		x	x
Reusability		x			x

While all these models try to capture the subjective concept "quality" for software source code, new quality models that capture the quality of models (i.e., CIMs, PIMs, or PSMs) from the viewpoint of architects, analysts, or maintainers are still missing.

2.2.2 Software development process and maturity models

Beside the problem-oriented approach of diagnosis quality defects many other approaches are known to improve the software development process and the resulting software quality. However, these process-oriented quality assurance techniques and quality defect diagnosis cannot be seen in isolation. Quality defect diagnosis have to be integrated into a software development process, such as for instance the Waterfall model, the Spiral model or model-driven development processes (described in (Vide, 2007a)) in the VIDE project. Quality assurance plays an important role in most of these process models. Independent from the software process model used it is important to understand the maturity of the software development and the quality standards archived.

A couple of frameworks have proposed to access the process maturity or an organisation or a project. Examples for process maturity frameworks are Capability Maturity Model Integration (CMMI) (SEI, 2006), Software Process Improvement and Capability Determination (SPICE) or ISO/IEC 9000-3 (ISO, 2005) the software specific variant of ISO 9000. Common to most process maturity frameworks is that the development process is evaluated and classified into *maturity levels*. Quality assurance and automatic defect detection on model level as described in this document supports organisation or project to increase the maturity level.

We'll use the CMMI to illustrate the benefit of defect detection for process maturity (for a short overview see http://en.wikipedia.org/wiki/Capability_Maturity_Model). CMMI utilizes five maturity levels that build on top of one another. These levels describe best practices that should be used by an organization. They are:

1. **Initial**: Initial state with no specific requirements.
2. **Managed**: Projects are managed and similar projects are successfully repeatable.
3. **Defined**: Projects are executed according to an (adapted) software development process which is improved over time.
4. **Quantitatively Managed**: The software development effort effectively controlled using statistical and other quantitative techniques, and is quantitatively predictable.

5. **Optimizing**: Continuous improving process performance towards quantitative objectives. The objectives are continually revised to reflect changing business objectives, and used as criteria in managing process improvement.

Since software quality assurance by diagnosing quality defects contributes to predictable improve product quality using statistical and other quantitative techniques the methods contributes partly to maturity level 2 (“Process and Product Quality Assurance”: diagnosing quality defects), level 3 (“Decision Analysis and Resolution”: handing and deciding about quality defects), level 4 (“Quantitative Project Management”: measurement & statistics about quality defects), and level 5 (“Causal Analysis and Resolution”: root cause analysis of quality defects and initiating preventive actions) (SEI, 2006).

2.3 Quality Defects and Quality Defect Diagnosis

The main concern of software quality assurance (SQA) is the efficient and effective development of large, reliable, and high-quality software systems. While verification and validation efforts in industry typically focus on functional aspects, using techniques such as testing or inspection, other quality aspects are often neglected. However, the non-functional quality of a software product is crucial for its evolution and maintenance by the same or another software developer. Other techniques as software product analysis and measurement are either used to measure software systems and interpret their quality based on a previously defined quality model or to predict project characteristics based on experiences from past measurements. From the deficits found by interpreting the quality characteristics (e.g., software metrics), further actions are derived on an abstract level to improve the software quality.

Another approach in SQA is the diagnosis of explicitly defined defects such as anti-patterns, design flaws, or code smells that represent system-independent defects with a negative effect on a quality such as maintainability. Individual refactorings are used to remove these defects and improve the defective parts without changing its functionality.

The techniques to diagnose quality defects (i.e., smells, antipatterns, flaws, etc.) are mainly based upon research from the field of software refactoring that is very active and beginning to address formalisms, processes, methods, and tools to make refactoring more consistent, plannable, scaleable, and flexible (Mens & Tourwe, 2004). As Bennett and Rajlich state in their roadmap paper, the central research problem is the inability to change software easily and quickly. Current research issues are being addressed by gathering more empirical information about the nature of software maintenance. The removal of unnecessary complexity is sought through the preservation and management of knowledge for future software maintenance and restructuring of code (Bennett & Rajlich, 2000).

2.3.1 Automated Quality defect diagnosis techniques

Currently, several tools were being developed that automatically support parts of the refactoring process. Some of these tools automate the realization of refactorings (e.g., “Extract Method”) – but the detection of places where to apply the refactoring (i.e., quality defects) is still a manual process. Several techniques were developed for code clone detection (Bruntink et al., 2004), obsolete parameters or inappropriate interfaces (Tourwe & Mens, 2003), and the general processing of source code for of diagnosis and visualization of code smells (van Emden & Moonen, 2002).

While some techniques for the diagnosis of quality defects are already known (e.g., the “long method” code smell or several “architectural smells” in the sotograph tool) techniques for several other quality defects are currently unknown. This is especially true for quality defects that are only diagnosable by analyzing several versions from a software repository.

2.3.2 Quality defect handling methods

In addition, the *handling of quality defects and removal activities* in the lifecycle of a software product are not well treated in the literature. For example, the ODC process consists of an opening and closing process for the 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.

Software annotation languages used in source code such as JavaDoc or Doxygen can be applied to document the functionality and structure of the software system at the code level. They are tailored for the automated generation of API documents based on a machine-readable syntax. The handling of potential quality defects is not addressed such that accepted quality defects are not presented over and over again and decisions are preserved. Language extensions or mechanisms for machine-readable storing of information about symptoms, defects, or treatments (change history) have not been published.

2.4 Software Quality Improvement Techniques

Software Inspections, and especially code inspections, are concerned with the process of manually inspecting software products in order to find potential ambiguities, 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 looked after is smaller.

Software Testing and debugging is concerned with the diagnosis of defects regarding the functionality and reliability as defined in a specification or unit test case in static and dynamic environments.

Software product metrics are used in software analysis to measure the complexity, cohesion, coupling, or other characteristics of the software product that are further analyzed and interpreted to estimate the effort of 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 sources for negative effects on the quality (e.g., the M-System, ZD-MIS, or the Sotograph). The automated tool-based detection of specific anomalies affecting the quality in software products is relatively rare, to non-existent. Most of these tools (like Checkstyle, FindBugs, Hammurapi, or PMD) analyze the source code of software systems to find violations against 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, 2006).

2.5 Beyond the State of the Art

Important parts of the work of the VIDE project contribute to the fields of refactoring, maintenance, and quality engineering for model-driven software development. The primary contributions to the practice and theory will be:

- A catalogue of existing and the definition of new techniques for quality defect diagnosis (i.e., deliverable D4.2). This includes techniques for the extraction, transformation, and integration of information from VIDE-based models to enable model-based quality defect diagnosis techniques.

- A formal model of quality defects on the PIM level that describes quality defects, their structure, symptoms, affected qualities, and associated refactorings as well as their interrelations and dependencies (i.e., this deliverable D4.1a). The model will be usable to classify new quality defects, diagnose quality defects based on identified symptoms, and to configure an optimal treatment (i.e., refactoring) plan.
- Development and evolution of a domain-specific quality defects model from the generic model of quality defects for the domain of business application.
- An extension of the VIDE platform (based on the eclipse-IDE) for the analysis of software models (to de developed in WP9). It will consists of a plug-in based architecture that is easily extended and adaptable to other modeling languages (with respect to VIDE language extensions), abstraction layers (e.g., other models in MDSD as the CIM), or versioning systems.

3 Description of Research Approach and Methodology

This systematic literature review is based upon the frameworks as described for software engineering by Kitchenham (Kitchenham, 2004), Biolchini et al. (Biolchini et al., 2005) and Mendes et al. (Mendes, 2005) as well as guidelines for medical research by White et al. (White & Schmidt, 2005), Pai et al. (Pai et al., 2004), and Khan et al. (Khan et al., 2001). A systematic literature review is a means of identifying, evaluating, and interpreting all available literature relevant to a particular research area. The goal of this review is to systematically elicit all available literature on quality defects and quality defect diagnosis techniques. It was used to help to reduce the influence of the reviewer's own bias and supports this by deciding in advance what evidence to use and how to use it, so these decisions are not influenced by the evidence itself.

Systematic literature reviews play a central role in the gathering and structuring of scientific knowledge. As science is a collective and cumulative endeavour, any theory, methodology, or technology is suspect of validity threats and must be supported by evidence, as hard as possible. Moreover, all too often new knowledge, techniques, and methods are proposed and introduced, without building on the existing body of knowledge. These problems can be somewhat alleviated by collecting and structuring the available body of knowledge using the mechanism of literature review. Systematic literature reviews help to make the implicit theories explicit by identifying their commonalities and differences, and may even be an impulse for the unification of existing theories to induce a new, more general theory.

This section describes the design of the systematic literature review in order to state the underlying goals and make it possible to easily replicate or extend this literature review later on.

3.1 Background and General Objectives

This review is targeted to help to improve the situation for quality defect diagnosis in software engineering in several ways. Firstly, as a means to summarize existing literature and construct an objective and comprehensive overview about quality defects, related concepts, and their diagnosis techniques. Secondly, to derive definitions of existing quality defect related concepts and synthesize a consistent and uniform definition of quality defects. Finally, to identify gaps in the current research and body of knowledge, this might be used to determine where future research is needed.

3.2 Review Method

In order to systematically conduct the review we based the research method on the process as defined by Barbara Kitchenham (Kitchenham, 2004). Therefore, the following phases were conducted to realize this literature review:

- *Background research*: Initial scoping survey to identify the need for a review as well as search terms for quality defects and their diagnosis techniques
- *Review planning*: Specification of the research question(s), required data, search terms, and identification of search engines (i.e., data sources). This resulted into a review protocol that is part of this section.
- *Identification of literature*: Searching for literature in the search engines and retrieving titles, abstracts, and reference material.
- *Selection of literature*: Reading of literature abstracts, including (i.e., selecting) and excluding literature, and obtaining full text versions of the selected literature. Analyzing of

the references in the obtained literature to identify further literature (i.e., repeat this phase with the new list of literature)

- *Quality Assessment*: Reading the full papers, evaluating appropriateness, and identification of bias.
- *Data Extraction*: Extraction of relevant data from the literature.
- *Data synthesis*: Structuring and systematization (descriptive / non-quantitative) of the quality defects and quality defect diagnosis techniques found.

The systematic literature review was conducted from July 2006 to June 2007 using the techniques described in the following subsections.

3.3 Review Questions

The review question or research aim of a systematic literature review focus on gathering and interpreting evidence, deciding on the cause of a problem, predicting a possible outcome, deciding on solutions to apply, or the determination of preventive measures. In software engineering additional foci might be added that are concerned with the classification of literature to a pre-defined model (e.g., as in (Laitenberger, 2002)) or the construction of an ontology as it is the goal of this review.

The research questions in this systematic literature review are targeted to support the construction of an ontology about quality defects and techniques for their diagnosis. This review is focussed to answer the *primary research question*:

> *Which quality defects exist and to which extent are they diagnosable via (semi-) automated techniques in the context of VIDE (i.e., especially behavioral and data-intense QDs in PIMs based on UML with action languages)?*

This question does not consists of the components *condition/disease* (i.e., the type or set of quality defects), *population/systems* (i.e., the investigated (type of) systems), *intervention/method* (i.e., the techniques itself), and *outcome/effect* (i.e., the effect of the intervention on the condition) as described in (White & Schmidt, 2005). The primary question is not very focused on a specific type of quality defect or diagnosis technique (e.g., as in "*What is the most efficient diagnosis technique to diagnose the 'Long Method' code smell in 10k-100k large object-oriented embedded software systems?*") as there is not enough literature available (based upon our knowledge from an initial scoping survey). In order to concretize the primary question following *secondary research questions* are given:

> *Which quality defects exists resp. are described in the literature and under which names are they known? (i.e., identifying existing quality defects and related concepts)*
>
> *How do the concepts for quality defects differ, what artifacts are affected, and where are gaps (i.e., missing defect description or diagnosis techniques)?*

3.4 Data Sources and search terms

The *search strategy* applied included the following *data sources* for identifying as much as possible of the relevant literature. The following journals, conferences, and workshops were investigated from December 2006 back to January 1990:

- The conferences and workshops used an information source were: International Conference on Software Engineering (ICSE), Working Conference on Reverse Engineering (WCRE), International Conference on Software Maintenance (ICSM), European Conference on Software Maintenance and Reengineering (CSMR), Technology of Object-Oriented Languages and Systems (TOOLS), European Conference on Object-Oriented Programming (ECOOP), Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), eXtreme Programming and Agile Processes in Software Engineering (XP), European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE), Software Metrics Symposium (Metrics), Symposium On Applied Computing (SAC), Symposium on Software Reliability Engineering (ISSRE), Aspect-Oriented Software Development (AOSD), Asia-Pacific Software Engineering Conference (APSEC), International Symposium on Empirical Software Engineering (ISESE), International Symposium on Software Testing and Analysis (ISSTA)
- The journals used as an information source were: Transactions on Software Engineering (TSE), IEEE Software (IS), Transactions on Software Engineering and Methodologies (TOSEM), Information and Software Technology (IST), Journal of Systems and Software (JSS), Software Practice and Experience (SPE), Software Testing Verification & Reliability (STVR), Software Quality Journal (SQJ), Journal of Software Measurement (JSM), Transactions on Architecture and Code Optimization (TACO), Automated Software Engineering (ASE), Empirical Software Engineering (ESE), International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Computing Surveys (CS), ACM, Software and Systems Modeling (SOSYM), Software Engineering Notes (SEN), and the Journal of Software Maintenance and Evolution (JSME).

Furthermore, the following *search engines* were used to browse through several conference proceedings and journals as well as to find publications from other sources. These search engines and the query language mechanics used to search in the titles, abstracts, and keywords are:

- *IEEE Xplore*: The search engine for the literature by IEEE provides full text access to the technical literature in computer science including many conferences and journals. Although this search engine is capable to searching in the full text we only searched in general in the appropriate metadata (i.e., document title, abstract, and index terms). In the case of searching specific journals and conferences we included the full text search. The query string, e.g., for the search term “code smell” had the form: “((smell<in>metadata) AND (code<in>metadata))”.
- *ACM Digital Library & The Guide*: This publication search engine by ACM enables the access to the collection of citations and full text from journals, conferences, and newsletter articles published by ACM and other publishers. In order to widen the results we searched in the full text of the publications. The query string, e.g., for the search term “code smell” had the form "code smell". In special cases where too many papers were found we constrained the search to the title and abstract using the query “(title:code title:smell) (abstract:code abstract:smell)”.
- *INSPEC* (via “fiz technik”): The access to the INSPEC bibliographic library for computer science by the company fiz technik. As the query interpretation is very strict, plurals of search terms had to be included, e.g., by using wildcards such as “code smel*”, and the search was constricted on the title and the sections “Computers and control” and “Information technology” of INSPEC.
- *OCLC FirstSearch* (incl. WorldCat, ECO, and ArticleFirst): The online computer library center includes many books, journals, and conferences. The query string, e.g., for the

search term “code smell” had the form: “ti:’code smell’ OR ti:’code smells’” in the keyword section (that includes “words from titles, subject headings, and notes”) .

- *Springer Link*: The search engine for all publications by springer including conferences, journals, and the LNCS as well as LNI series. In order to widen the results we searched in the full text of the publications. The query string, e.g., for the search term “code smell” had the form “code smell” stated as a phrase via the “advanced search” form.
- *DBLP*: The digital bibliography & library project by the University of Trier, Germany, provides access to computer science bibliographies of conferences, journal, and individual persons. The search in this database was conducted on the titles of the publications using, e.g., the term “title = "code smell"“ (via Advanced Search) to search for “code smell”.
- *Citeseer*: Another public bibliographic search engine and digital library like DBLP that is hosted by the Pennsylvania State University, USA. We used the standard search available that searches in the full text of the indexed publications. The query string, e.g., for the search term “code smell” had the form "code smell or (code and smell)".
- *Google Scholar*: The internet-based search engine for online publications by Google is used to diagnose either grey literature or publications and tools not listed in the commercial indexing services above. As this search engines searches in the full text of publications, additional literature was found that did not include the search terms in their title, abstract or keyword list. The search term was used as a simple phrase, e.g., “code smell”, constrained on the section “Engineering, Computer Science, and Mathematics”.
- *Amazon.com*: The online book store was used to search for relevant books using the title and subject search in the “advanced search” feature of the books section. A large amount of irrelevant literature was reduced by focusing the search on the category “computers & internet” and then “programming” or “computer science”. The search term was used as a simple phrase, e.g., “code smell”.
- *A9.com*: Amazons full text book index was used to search for relevant books and chapters that included a search term. The search term was used as a simple phrase, e.g., “code smell”.
- *Google Book Search*: The book search engine provided by Google was used as A9 to find relevant books and chapters that included a search term. The search term was used as a simple phrase, e.g., “code smell”.
- *Google Internet search*: The global internet search engine by Google was used to find technical reports, dissertations, etc. on the internet. While the index of the internet in the search engine is not complete it is the best fit to search on the internet. The advanced search capabilities were used to find full documents in PDF format containing the search terms by using the query “filetype:pdf +<type> +<search term>” where “<type>” was replaced by the type (e.g., “dissertation”) and “<search term>” was replaced by the individual search term in phrase form (i.e., including quotations).

Finally, all relevant references cited in the selected publications (i.e., after the step described in section 3.5) and the publication lists of the authors (using the DBLP author search) were used to find additional literature.

In order to constrain the search only English literature was included in the review – even when it was known that, for example, German literature on this topic was available. We excluded non-English literature as it is the main scientific language (i.e., every SE scientist can understand its content). Nevertheless, to retrieve as much as possible of relevant literature from these search engines several synonymous *search terms* were used:

- The search terms for the retrieval of quality defect related literature included “code smell”, “bad smell”, “design flaw”, “antipattern”, “anti-pattern”, “antipractice”, “anti-practice”, “antiidiom”, “anti-idiom”, “design heuristic”, “design characteristic”, “design defect”, “pitfall”, “cliché”, “bug pattern”, “defect pattern”, “refactoring opportunity”, “anomaly”, and “quality defect”. Furthermore, the very similar concepts “code style”, “coding style”, ”code convention”, “coding convention”, “code rule”, and “coding rule” were used.

These terms are based on knowledge acquired during previous unsystematic literature survey and refined resp. extended during the initial scoping survey of the systematic literature review.

3.5 Literature Selection and Literature Quality Assessment

The results from the literature collection (i.e., references to the papers) were then manually read to identify and *select* relevant literature. Unfortunately, the literature on quality defect diagnosis techniques is not always based on hard evidence and, therefore, no further quality standard (e.g., requiring a controlled experiment in industry) were applied to filter the literature except that it had to include a quality defect, definition, taxonomy, or diagnosis technique.

However, much information is available on quality assurance techniques it has not been easy to reconcile and consolidate information on quality defects due to the sheer volume of work already available. In order to focus and sharpen the literature survey we *included* all literature matching the abovementioned search terms for quality defects but *excluded* the following quality defect related concepts:

- Functional defects (i.e., errors detected by testing an executable (part of) a system)
- Performance characteristics (i.e., failures to process in time or to process a heavy workload (e.g., many users) by testing an executable (part of) a system)
- Specific or non-abstract defects (i.e. specific to a software system)
- Pitfalls in form of case studies of projects, etc.
- Law, finance, procurement, and marketing related pitfalls, etc.
- Books with less than 5 pages about a concept, no described concept (instances)
- Literature solely about refactoring and “indirect” refactoring rationals (i.e., without explicit descriptions of smells or other refactoring opportunities).

Furthermore, we excluded articles based on the following rules: (a) it takes a considerable effort (money or time) to get the article and (b) duplicate publications will be identified by cross-checking authors and diagnosis technique. Finally, as we do not synthesis a quantitative statement from the literature we do not suspect publications to be invalid per se and, therefore, did not reject grey (non-peer-reviewed) literature such as PhD theses or technical reports.

3.6 Data Extraction

In order to answer the primary and secondary research questions, data has to be extracted from the identified and selected literature. Based upon the recommendations in (White & Schmidt, 2005) and the research questions we extract the following data for the *quality defect* related literature:

- *Reference information*: The author names and date of publication.

- *Name for the quality defect concept*: The term or phrase used to name the quality defects (e.g., code smell, bug pattern, or aspect smell).
- *Formality of the description*: The descriptions of the quality defects will be categorized in the three categories informal, semi-formal, and formal. Informal for unstructured plain-text descriptions, semi-formal for structured but potentially ambiguous text passages (e.g., sections for name, symptoms, …), and formal for unambiguous representations (e.g., in first-order logic).
- *Number of quality defects described*: The amount of distinct quality defects described in this publication for a specific artifact type.
- *Definition of the quality defects*: The definition of the quality defect used in this publication.
- *List of the quality defects*: The names of the quality defects.
- *Description of the quality defects*: The description of the quality defect.
- *Design entity involved*: The design entities involved in the quality defect (e.g., classes or inheritance relations).
- *Quality affected*: The quality aspect of the artifact influenced by the quality defects (e.g., maintainability of source code or the performance of a process).

3.7 Data Synthesis Activities

The objectives of the descriptive or non-quantitative synthesis (Khan et al., 2001) is the collection and unification of the terminology for quality defects and quality defect diagnosis techniques. Key elements of the synthesis are typical names for quality defects, commonalities of techniques (e.g., used metrics), similarities of the evaluation contexts, and the results of the evaluations. The synthesis might indicate the absence of quality defects of a specific type or diagnosis techniques for specific quality defects. Furthermore, it might demonstrate the heterogeneity (i.e., variability) or homogeneity (i.e., similarity) of the diagnosis techniques in terms of key characteristics, quality of the diagnosis, or effects.

The *characterization of quality defects* was build upon the analysis of the quality defects described in the literature. First a list of the different types of quality defects, their definition, and their (structured) templates were collected. Second the artifacts the quality defects appear in, the (potential) quality aspects they affect, and the type of facet of the artifact they describe (e.g., dynamic behavior) were identified.

The results of this literature survey are presented in the next section.

4 Quality Defects and Related Concepts

Successful MDA is expected to make the models the main development artifacts, replacing today's programming languages analogous to the way high level programming languages have previously replaced assembly languages (Mellor et al., 2004). When moving to a completely model based development approach the quality of the models from which the applications are generated becomes very important. In order to assist the modeler of a PIM during his work information about possible threats to the quality (in respect to ISO 9126) of the PIM should be indicated as early as possible. While the research on intelligent assistance in software engineering started in the 1970th the maturity and integration of these techniques is lingering but demanded by software engineers (Rech et al., 2007).

In this work package quality defects such as architectural smells, anti-patterns, and design flaws are investigated explore new quality defects that might occur on the model level were investigated. Quality defects often stem from experiences made by practitioners and consultants in different software projects, domains, and environments. However, other techniques for the extraction of these recurring problems exist such as knowledge discovery in databases (Rech, 2004) or semi-automated techniques based on experience factories (Rech & Ras, 2007, in work).

In VIDE, quality assurance knowledge for platform-independent models will be researched to increase their quality and ease the development and maintenance of these models. The knowledge explored in this work package will be used to develop a module of VIDE in WP 9 that discovers quality defects from the PIM and annotates its textual and visual representation Furthermore, it will be used to enrich the visualization of the models in order to inform the designers and maintainers about potential threats to model quality.

While some techniques for the discovery of quality defects in source code are already known, the discovery of quality defects based on architectural information in early development phases, such as design, are not well understood and open to further investigation. With the rise of MDA the need for high-quality and maintainable software models will increase.

The first section gives an overview of quality defects and other information discovered by the systematic literature review. The following section will go into more detail and list the quality defects found grouped into the concepts they were described under.

4.1 Overview & Visualization of Concepts

Publications including comprehensive overviews about quality defects as well as *classifications, taxonomies, ontologies, or templates* of quality defects are very rare. Typically, classifications are used in books for collections of refactorings (Fowler, 1999), code smells (Wake, 2003) (Mäntylä et al., 2003), anti-patterns (Brown et al., 1998), design flaws (Riel, 1996a), design characteristics (Whitmire, 1997), or bug patterns (Allen, 2002) as well as reengineering patterns (Demeyer et al., 2003). They all define proprietary and different formats for the description of quality defects that are not compatible among each other and neglect information about affected software qualities. There is no comprehensive taxonomy, ontology, or model that helps to classify and distinguish quality defects, their symptoms, and treatments in a uniform way (i.e., similar to the taxonomies in medicine or biology).

Defect classification schemes (Freimut, 2001) used in software measurement and testing such as ODC are not designed to describe quality defects in a formal, consistent, and complete

way. They are designed to support the defect documentation and management and help in the reporting about the software quality, the planning and tailoring of future quality improvement activities (e.g., test planning), and the initiation of preventive measures in early development phases.

4.1.1 Literature corpora overview

During the literature review as described in section 3 we extracted publications including predefined search-terms from the body of software engineering literature. As depicted in Figure 1 we found 560 publications relevant to our topic that included either information on quality defects or their diagnosis techniques. These findings included 61 books, 35 theses, 131 journal paper, 308 workshop and conference articles, as well as 25 reports, chapters, and webpages.

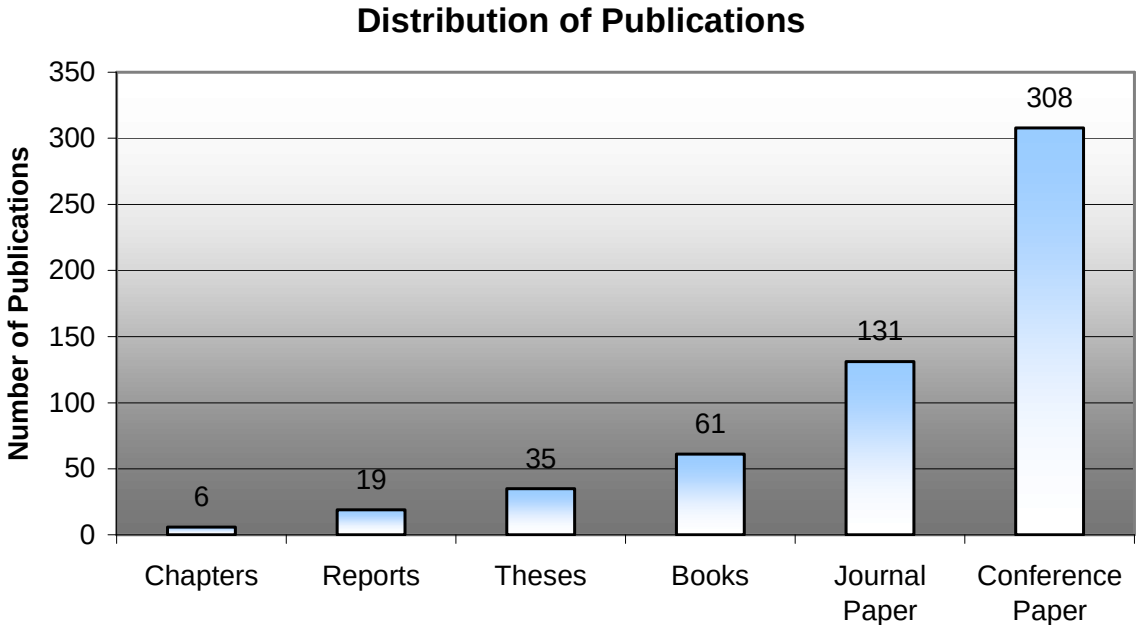

Figure 1. *Literature Type about Quality Defects*

From this corpora of knowledge the main source for quality defects are books and (PhD) theses. Typically, these publications list groups of quality defects relevant to one abstraction level (e.g., design, test, or code) or quality aspect (e.g., performance antipatterns). However, some of them make an all around sweep and present quality defects on multiple levels (e.g., management, coding, and reuse pitfalls).

Nevertheless, most quality defects are described informal and therefore problems arise as it is not clear how to (best) refactor or treat them. A systematic and empirical investigation of these quality defects – and especially their impact on the software quality – is advised.

The largest groups of publications are, as expected, workshop and conference papers. As presented in Figure 2 a more systematic analysis and presentations of quality defects and their diagnosis techniques can be found in conferences such as ICSE (International Conference on Software Engineering), CSMR (European Conference on Software Maintenance and Reengineering), and ESEC/FSE (European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering).

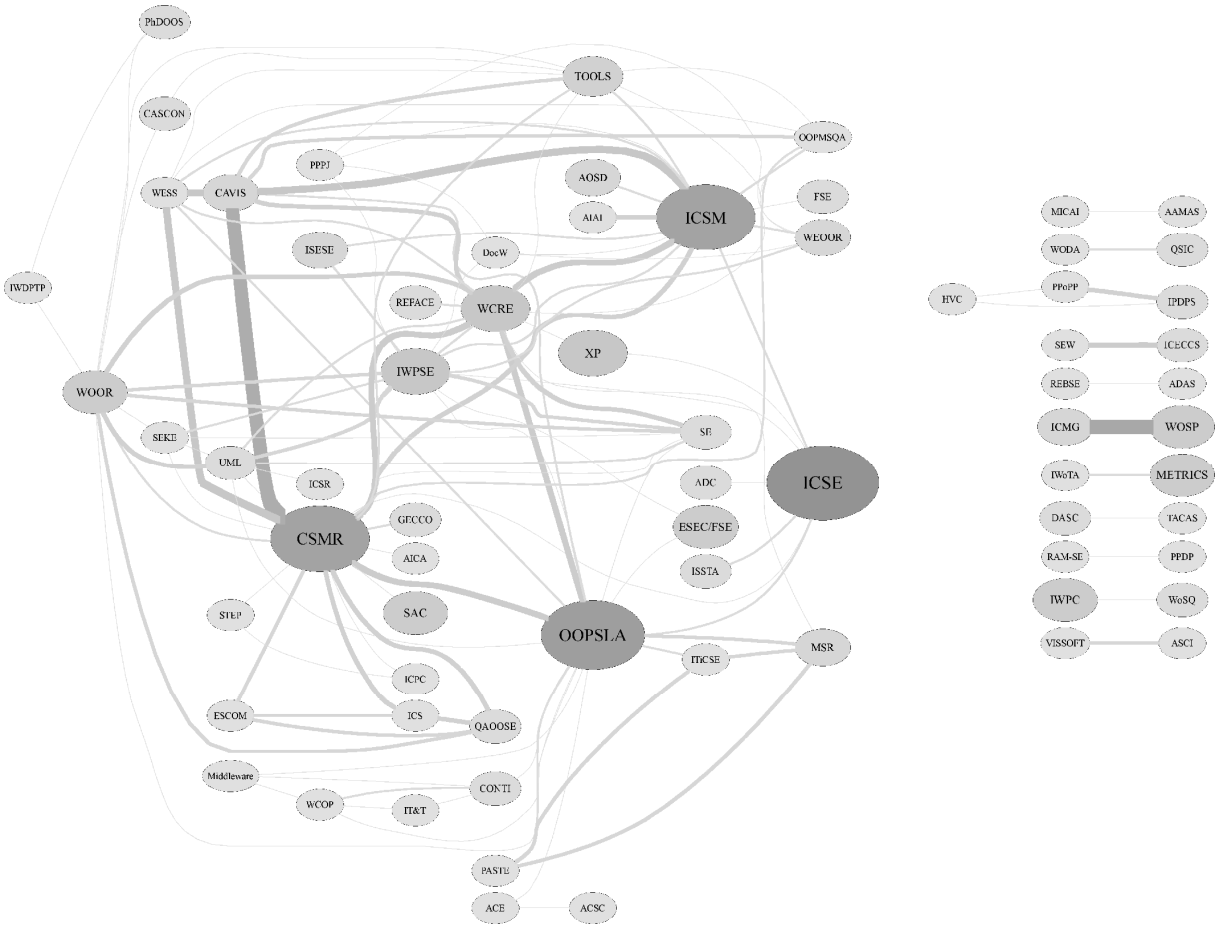

Figure 2. *Conferences with contributions about quality defects*

However, the co-publication analysis (i.e., represented by the lines – the thicker the more authors presented at both conferences) shows that many workshops and conferences are not attended by the same communities. For example, while ICSE and CSMR are both conferences where many papers are published only few authors have papers (related to quality defects or their diagnosis) on both conferences. But many authors publishing on the CSMR conference do publish on CAVIS workshop.

Nevertheless, in all these communities and sub-communities many different names are used to name these groups of quality defects. But are there really characteristics that can be used to differentiate between them? And what types of quality defects are described in the literature? Under which names are quality defects known and where do they differ? We observed that a common terminology (Naming) could not be found – author are typically using striking names for the defects.

In summary, we discovered approx. 800 quality defects in these larger collections alone. While several other publications such as conference papers, reports, etc. additionally list several other quality defects that are not described in these collections this number can be seen as a first rough estimation. Furthermore, some quality defects described in one collection reoccur in other collections under different names.

4.1.2 Available Information Structures for Quality Defects

Most larger collections of quality defects such as antipatterns, pitfalls, code smells, etc. use a semi-structured template to present the individual quality defects. The Information encoded in these structures has to be used to decide upon the applicability of these quality defects in our

given context (i.e., if they satisfy the given criteria). The core criteria for the identified quality defects were **system-independence**, i.e., the described problem have to be independent from the specific software system and its requirements.

Table 2 lists and compares the existing template formats used for different quality defects in larger collections. As values for the criteria "●" was used to denote that a special slot exists in the template and the content satisfies the requirements. The "❍" symbol was used to denote that a slot does exists but the information is not sufficient for the requirements and "○" means that a slot does not exist, the information is not sufficient for the requirements, but the information is available in more than half of the defects. Finally, the symbol "-" denotes that information is not or only sparely available (less than half of the defects). In this comparison the following attributes and criteria were used:

- **Formality**: Captures the degree of formality the template adheres to. *Informal* templates refer to single free-text blocks were quality defects are described solely in prose. If the quality defect description is partitioned into several section with a specific focus (e.g., causes, treatments, forces, etc.) they are classified as *semi-formal*. *Formal* representations allow reasoning by machines and are fully unambiguous (e.g., by using OWL or first-order predicate logic).
- **Name**: A clear and precise name that communicates the problem and is based on a structured taxonomy (i.e., similar problem should be named in a similar way).
- **Description**: A unambiguous description of the core problem – eventually split in several more specific sections (e.g., "anecdotal evidence")
- **Interrelation**: Captures if relations to other defects are described.
- **Causes**: Captures if the causes for the quality defects are explained or referenced.
- **Treatment**: Captures if direct treatments (e.g., refactorings) are described to remove or attenuate the defect.
- **Effects**: Captures if effects of the defects on the quality aspects are covered.
- **Symptoms**: Captures if identifiable characteristics (e.g., metrics) are stated that can be used in the diagnosis.
- **Diagnosis**: Captures if techniques, thresholds or other means, that support the automated diagnosis, are given.
- **Indication**: Captures if techniques or guidelines are given to decide on the treatment in a given context.
- **RCA**: Captures if techniques are stated to identify or analyze the root causes of this defect.
- **Contra-diagnosis**: Captures if information is given to decide or change if the diagnosis is applicable in a specific situation.
- **Preventions**: Captures if techniques or guidelines are given to prevent this defect to emerge.
- **Principle**: Captures if the underlying principle (or anti-principle) is stated that caused the defect.

Table 2. *Information Content of QD templates used in larger collections*

QD	Source(s)	Formality	Name	Description	Interrela-tion	Causes	Treatment	Effects	Symptoms	Diagnosis	Indication	RCA	Contra-diagnosis	Preven-tions	Principle
Smells	Fowler, Kerievsky	Informal	O	●	-	-	○	-	-	-	○	-	-		-
	Wake	Semi-formal	O	●	-	●	●	O	O	○	-	-	O		-
	Rook & Lippert	Informal	O	●	-	-	-	-	○	○	-	-	-		-
Anti-patterns	Brown	Semi-formal	O	●	-	●	O	O	O	○	-	-	O		-
	Dudney	Semi-formal	O	●	-	●	●	O	O	○	-	-	O		-
	Tate	Semi-formal	O	●	-	O	O	O	O	-	-	-	-		-
Heuristics	Riel	Informal	O	●	-	-	○	-	-	○	-	-	-	-	-
	Gibbon	Informal	O	●	●	-	O	-	-	-	O	-	-	-	○
	Grotehen	Semi-formal	O	●	●		O	●		●			O	O	-
	Frater	Informal	O	-	-	-	-	-	O	-	-	-	-	-	-
Pitfalls	Webster	Semi- formal	O	●	-	-	O	O	O	O	-	-	-	O	-
	Daconta	Informal	O	●	-	-	○	-	-	-	-	-	-	-	-
Other	Marinescu (Flaws)	Semi-formal	O	●	-	-	-	○	●	●	-	-	-		-
	Marinescu Lanza (Flaws)	Semi-formal	O	●	○	-	O	○	○	●	-	-	-		-
	Allen (Bug Pat-terns)	Semi-formal	O	●	-	O	O	-	O	-	-	-	-	O	-
	Bloch (Puzzles)	Informal	O	●	-	-	○	-	○	-	-	-	-	-	-
	Johnson & Foote (Rules)	Informal	O	●	-	-	○	-	○	○	-	-	-	-	-
	Robbins (Critics)	Semi-formal	O	●	-	-	O	O	○	●	-	-	-	-	-
	Telles/Hsieh (Bugs, Errors)	Semi-formal	O	●	-	-	-	O	O	-	-	-	-	-	-
	Younessi (Design Defects)	Informal	O	●	-	-	-	-	-	-	-	-	-	-	-
	Visaggio (Ageing Symptoms)	Informal	O	●	-	-	-	-	-	-	-	-	-	-	-
	Hawkins (Illnesses)	Semi-formal	O	●	O	-	O	-	O	-	-	-	-	O	-
	Lorenz & Kidd (Thresholds)	Semi- formal	-	-	○	-	O	-	O	O	-	-	-	-	-

As we can see most quality defects are described in semi-formal templates with varying grades of information content. True formal templates are currently not developed.

The classifications described in this section can be used on all discovered quality defects. However, a comprehensive classification that satisfies all previously stated criteria is still missing.

4.1.3 Comments to the following collection

The literature survey was used to extract data as described in section 3.6 that can be used to identify Quality Defects. In order to reduce the available information the following requirements were applied to filter the information stated for the quality defects. These requirements, stated in decreasing priority, are:

6. The quality defects should be applicable on the PIM level in MDSD.
7. The quality defects should have a relation to data-intense software systems.
8. The quality defects should focus on behavioral aspects of the system.
9. The quality defects should be visualizable in a single (local) diagram (i.e., no multi-diagram or distributed defects such as the code smell "Shotgun Surgery")

In the following lists of quality defects we will denote the individual defect with a "●" if it is fully applicable, with a "❍" if it is partial applicable, with "○" if it is irrelevant or counter-productive (e.g., multi-location defects).

The final selection of quality defects that are targeted with diagnosis techniques (in D4.2) and that build the basis for the diagnosis tools (in WP9) are described in section 6. Furthermore the following information is included within the tables:

- **Name**: The original name of the quality defect as described in the source or a new name based on the description.
- **Type of Quality Defect**: A rough classification of the quality defects into structural (System composition), semantical (Name/Identifier based), behavioral (Control flow / statement based), historic (System evolution based, e.g., using CVS, SVN, …), communicative (Message based), or layout (diagram based) quality defects. The type describes the main source of information that can be used to diagnose the problem (rule-based, not necessarily statistical).
- **Design Entities involved**: Larger entities involved in the quality defect – additionally the required information from the main source of information such as classes, methods, parameters (method), attributes (class), notes, statements (method body), versions, calls (method body or associations), etc.. The design entities do also indicate the information required to diagnose the quality defects.
- **Quality Aspects affected**: Only top level aspects from ISO 9126 are used – functionality, reliability, usability, efficiency, maintainability, and portability. Additionally, the aspects compilability and conformance are used. As no empirical data is available to support the effects a concretization to sub-characteristics of ISO 9126 (or another quality framework) was not pursued.
- **Description**: Short explanation of the problem.

Most of the 43 concepts are used by more than one author and comprises of several individual problems. This report summarizes information on the 22 concepts that are used in more than one publication or that comprises of more than ten individual quality defects.

However, there are 21 more concepts that are either used only by very few authors or comprises of very few problems and several terms such as "design problem", "design error", "design fault", "design failure", "design malfunction", "design degradation", or "design deficiency" were to general and did not result in any information on quality defect collections. The following list of concepts include term that are often used but are a) rarely used (i.e., only by one author), b) do not have many quality defects, or c) are on another level than software design or architecture (e.g., requirements analysis):

- **Refactoring candidates** (Kataoka et al., 2001) or **Refactoring Opportunities** *(Melton & Tempero, 2006)* (Tourwé & Mens, 2003) are synonyms to smells and are associated with one or more specific refactorings.
- **Design Disharmonies** is a concept used as an umbrella term similar to design flaws (Marinescu & Lanza, 2006)
- **Design Pattern Defects** are used to describe recurring errors in the design of a software that come from the absence or the bad use of design patterns. (Moha et al., 2005)
- **Design mistakes** is used rarely for development level problems (Becker, 2000a, 2000b)
- **Dysfunctional patterns** or **bad patterns** are a kind of anti-patterns, however, they represent "good" software patterns that are applied in a wrong context (Buschmann et al., 2007)
- Errors and other quality defects with a focus on security are also described as **Vulnerabilities** (e.g., used in (Livshits & Lam, 2005))
- Puzzlers are also described as **Traps**, Pitfalls, and **Corner-Cases** (Bloch & Gafter, 2005)
- The concept **fallacies** is used to describe worst-practices (i.e., anti-patterns) in the software engineering discipline but not on the level of source code or models (Glass, 2003)
- The medicine-based term **software cancer** was used to describe problems on the management level (Boundy, 1993)
- **Clichés** (e.g., standard algorithmic fragments or code snippets – such as searching algorithms, sorting algorithms, various data structures for representing sets, etc.) were used in knowledge-based software development environments (Waters, 1994). In general, they can be seen as a kind of precursors to software patterns.
- **Pratfalls** is a term sometimes used in conjunction with pitfalls (Wooldridge & Jennings, 1999)
- The concept Design **Problem** is used by (Munro, 2005) for problems similar to smells and flaws.
- **Bad design decisions** is used in conjunction with smells – especially if these decisions do occur in multiple systems.
- The term **Anti-idioms** is used for problems such as "NotWithin" (Schmidmeier, 2004) or "DoubleCheckedLockingIsBroken" (c2.com).
- **Anti-practices** are basically process-oriented antipatterns (Kuranuki & Hiranabe, 2004)
- **Inconsistencies** is also a term used in conjunction with Rules (Liu et al., 2002)
- OOD **Criteria** (Coad & Edward, 1993) and OO Goodness Criteria (Yourdon, 1993) are used for general guidelines such as minimize coupling, maximize cohesion, etc.
- Before design patterns became a hype and kind of standardized Tom Love used the concept OOD "**patterns**" (Love, 1991) (Yourdon, 1993) (page 310) such as "Objects should not access data defined in their superclasses"

Furthermore, several quality defects we found are platform-specific problems that appear on the first impression as irrelevant to the platform-independent level. However, as models on the PIM level are going to be transformed to the PSM level these problems should be taken into consideration either while modeling the PIM or in the development of PIM to PSM transformers. Being system-independent the consideration of these problems in general-purpose transformers or quality-checking transformers (i.e., on the PSM level) seems better in order to not overload the PIM level (that should not consider all platform-specific quality defects, e.g.,

for Ada or Cobol). Therefore, we integrated platform-specific quality defect concepts in this report but did not describe every single defect.

Additionally, every missing design or architectural pattern (or style) might be described as an "Absence of <Pattern>" quality defect (e.g., "Absence of Strategy" or "Absence of MVC"). While some of these problems are describe under one concept or another a comprehensive collection of all of the two thousand (Booch, 2007) architecture and design patterns is still missing (and would require techniques for the identification of design pattern candidates in a given context).

4.2 Ageing Symptoms

The concept "ageing symptoms" was used by Guiseppe Vissagio (Visaggio, 2001) to represent problems of a software system during its evolution (i.e., aging). In general, ageing symptoms are problems that are associated with one or more metric (i.e., concrete symptom) in order to identify points during the monitoring where the system starts to degrade. In the literature they are defined as follows:

- *"Each [aging symtom] is specified by metrics and the results of the measurements made suggest what operations should be undertaken to renew the software"* (Visaggio, 2001)

In the following sections we will list most of these ageing symptoms that were found in the literature survey. The first large collection of ageing symptoms were collected by Guiseppe Vissagio (Visaggio, 2001).

Table 3. *Ageing symptoms by (Visaggio, 2001)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Pollution	Dynamic, Structural	Calls	Maintainability	Parts of the software system do not serve to realize functionality exploited by the users.	⊙	○	●	⊙
Duplicate programs	Semantic	Methods, Statements	Maintainability, Reliability	Identical source code	●	○	●	○
Obsolete programs.	Structure	Build Info, Calls	Maintainability	Programs that have source code but no corresponding executable.	○	○	⊙	●
Sourceless programs.	Structure	Build Info, Calls	Maintainability	Executable programs that have no source code associated.	○	○	●	○
Useless components.	Data	Data Access	Maintainability	Component produces or modifies useless reports (i.e., data, files, …)	●	●	●	●
Dead data.	Data	Data Access, Attributes	Maintainability	Variables created but not used by any component.	●	⊙	⊙	●
Dead code.	Control	Statements, Calls	Maintainability	Statements that cannot be reached by the control flow.	●	○	⊙	●
Embedded knowledge	Semantic	Methods, Statements, Names	Maintainability	Knowledge about the system and domain is spread over the whole system	●	○	⊙	○
Incomprehensible data and modules.	Semantic	Docu, Names	Maintainability	Variables or modules whose meaning cannot be understood from the documentation.	●	○	⊙	○
Missing capacities.	Structure	Functionality, Methods	Maintainability	Functionality that cannot be precisely localized in the software components	●	○	⊙	○
Poor lexicon	Semantic	Names	Maintainability	The name has only little lexical meaning or does not communicate	●	⊙	⊙	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
				the real meaning/intent.				
Inconsistent data and module names.	Semantic	Names	Maintainability	Data or modules whose name does not express their meaning.	●	⭘	⭘	●
Coupling	Structure, Control	Calls, Inheritance	Maintainability	Parts are linked by an extensive network of data or control flows.	●	⭘	⭘	●
Pathological files.	Structure	Data Access	Maintainability	Files created or modifies by different programs	●	⭘	⭘	⭘
Control data.	Structure	Statements, Data Access	Maintainability	Data that create communication among components	●	●	⭘	⭘
Module complexity.	Structure	Statements	Maintainability	Complexity by too many (algorithmic or procedural) if-statements	●	○	●	●
Layered architectures	Structure	Architecture, Calls	Maintainability	Architecture consists of different solutions that can no longer be separated	●	○	⭘	⭘
Useless Files.	Data	Statements, Data Access	Maintainability	A file not used or used by an useless program	●	●	⭘	⭘
Obsolete files.	Data	Statements, Data Access	Maintainability	The software uses a file but does not create new records.	●	●	⭘	⭘
Temporary files.	Data	Statements, Data Access	Maintainability	A temporary file is created, read but not updated and deleted by the system.	●	●	⭘	⭘
Permanent files.	Data	Statements, Data Access	Maintainability	A file that is created, used, modified but not cancelled (i.e., deleted)	●	●	⭘	⭘
Anomalous files.	Data	Statements, Data Access	Maintainability	The records of the file are not created but read, modified, and cancelled.	●	●	⭘	⭘
Semantic redundant data.	Semantic	Names	Maintainability	Variables or data with synonymous meaning or a “parent-child” inclusion	⭘	●	⭘	○
Computational redundant data.	Data	Data Access	Maintainability	Datum A can be calculated using other, available data (e.g. A = f(B,C))	⭘	●	⭘	○
Structure data.	Data	Data	Maintainability	Data has no connection to the domain but supports the DB structure (e.g., checksums)	⭘	●	○	○
Superimposed data structure.	Data	Statements, Data Access	Maintainability	Data structures that share the same address space	⭘	●	○	○

4.3 Anomalies

A concept that origin from a general term is the “anomaly” concept. The term was used in a IEEE standard (IEEE-1044, 1995) to describe anomal effects in a software system. Furthermore, the term is often used to describe unspecific situations in a software analysis (e.g., outlier). However, the term was additionally used in many other publications to describe concrete system-independent problems. These anomalies represent problematic parts of the software system that seem wrong, complicated, or cumbersome to an experienced developer. In general, anomalies are problems that are associated with one or more specific refactorings (i.e., concrete treatments) that might be applied to remove the anomalies. In the literature they are defined as follows:

- *“An anomaly is any condition that departs from the expected.” (IEEE-1044, 1995)*
- *“[anomalies are] properties inherent in implausible programs”(Kasyanov, 2001)*

- *"… anomalies which are symptomatic of programming errors" (Taylor & Osterweil, 1980)*

Beside the anomalies on the code or design levels many other problems were described using the anomaly metaphor. Today, we have anomalies on different abstraction layers, for development phases, or technologies such as concurrent software (Taylor & Osterweil, 1980), distributed systems (Cheung & Kramer, 1993), or knowledge bases (Baumeister et al., 2004).

In the following sections we will list most of these anomalies that were found in the literature survey. The first large collection of anomalies were collected by Kasyanov:

Table 4. *Anomalies by (Kasyanov, 2001)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Non-initialized variables.	Control	Statements	Maintainability, Reliability	An "information incomplete" execution, having the property that the value of a variable is referred to before any assignment to that variable.	●	○	●	●
Infinite execution.	Control	Statements	Maintainability, Reliability	A program is unable to pass through some of its points in its finite executions.	●	○	●	●
Useless objects.	Control	Statements	Maintainability	A variable (or procedure, mode and so on) has an explicit declaration but no uses, or a statement belongs to none of the program executions.	●	○	●	●
Redundant actions	Control	Statements	Maintainability, Reliability	A given program contains a statement that does not affect the results of all program executions.	●	○	●	●
Nonnatural constructions	Control	Statements	Maintainability	Some language construction used in a given program is more universal and/or complicated than the program actions represented by this construction.	⊙	○	●	●
Conflicting executions	Control	Statements	Maintainability, Reliability	Results of some collaterally evaluated fragments can depend on the way in which their evaluations are merged.	●	○	●	○
Semantically inadmissible or undefined constructions	Control	Statements	Maintainability, Reliability	An execution in which an index does not lie within the bounds of an array, illegal recursion, illegal side-effects, etc.	●	○	●	●
Absolute implausibility	Control	Statements	Maintainability, Reliability, Functionality	A program is called an absolutely implausible one if it has only meaningless executions (i.e. it has no executions without anomalies).	●	○	●	●

Table 5. *Concurrent Anomalies by (Taylor & Osterweil, 1980)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Referencing an uninitialized variable.	Control	Attributes, Statements	Maintainability, Reliability	An execution during which an event sequence of the form "purp" (arbitrary Program, Undefine, Reference, arbi-	●	○	⊙	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
				trary Program) for some program variable.				
A dead definition of a variable.	Control	Attributes, Statements	Maintainability, Efficiency	An execution during which an event sequence the form “pddp” (arbitrary Program, Define, Define, arbitrary Program) for some variable.	●	○	⭘	●
Waiting for an unscheduled process.	Control	Attributes, Statements	Maintainability, Efficiency, Reliability	This anomaly is represented by the event expression “puwp” (arbitrary Program, Undefine, Wait, arbitrary Program)	●	○	●	●
Scheduling a process in parallel with itself.	Control	Attributes, Statements	Maintainability, Efficiency, Reliability	This anomaly is represented by the event expression "pssp" (arbitrary Program, Schedule event, Schedule event, arbitrary Program)	●	○	●	⭘
Waiting for a process guaranteed to have previously terminated.	Control	Attributes, Statements	Maintainability, Efficiency	The expression “pwwp” (arbitrary Program, Wait, Wait, arbitrary Program) is symptomatic of this condition.	●	○	●	⭘
Referencing a variable which is being defined by a parallel process.	Control	Attributes, Statements	Maintainability, Reliability	For some variable both the event sequence “ps_0rdp” (P, S, Reference, Define, P) and the event sequence “ps_0drp” (P, S, Define, Reference, P) are possible.	⭘	○	●	●
Referencing a variable whose value is indeterminate.	Control	Attributes, Statements	Maintainability, Reliability	There exists a wait w_0 and two separate definition points for a given variable, d_1 and d_2, such that both the event expressions “$pd_1d_2w_0r$” and "$pd_2d_1w_0r$" are possible.	⭘	○	●	●

4.4 Anti-guidelines

Corrupt guidelines were used by Roedy Green in his essay “How To Write Unmaintainable Code“ to describe how (not) to write good code (Green, 1996). We call the guidelines for unmaintainable code simply “anti-guidelines”. These anti-guidelines represent problematic naming, comments, etc. in the software system that are misleading, wrong, complicated, or cumbersome to a developer or maintainer. In the literature they are defined as follows:

- *“[anti-guidelines are] tips … on how to write code that is so difficult to maintain …” (Green, 1996)*

As many of these anti-guidelines are similar (i.e., naming problems) or platform-specific only an excerpt of the 193 documented anti-guidelines in (Green, 1996) is given in the following table:

Table 6. *Anti-Guidelines for Unmaintainable Code by (Green, 1996)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
A.C.R.O.N.Y.M.S.	Semantic	Names	Maintainability	Use acronyms to keep the code terse. Real men never define acronyms; they understand them geneti-	●	⭘	⭘	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
				cally.				
Reuse Names	Semantic	Names	Maintainability	Wherever the rules of the language permit, give classes, constructors, methods, member variables, parameters and local variables the same names.	●	○	○	●
Recycle Your Variables	Semantic	Names	Maintainability	Wherever scope rules permit, reuse existing unrelated variable names.	●	○	○	●
Code That Masquerades As Comments and Vice Versa	Semantic	Names	Maintainability	Include sections of code that is commented out but at first glance does not appear to be.	●	○	○	●
Code Names Must Not Match Screen Names	Semantic	Names	Maintainability	Choose your variable names to have absolutely no relation to the labels used when such variables are displayed on the screen.	●	○	○	●
Document How Not Why	Semantic	Names	Maintainability	Document only the details of what a program does, not what it is attempting to accomplish.	●	○	○	●

4.5 Anti-patterns

In the nineties of the last century a new concept was transferred from architecture to computer science that helped to represent typical and reoccurring patterns of good and bad software architectures. These design patterns (Gamma et al., 1994) were the start of the description of many patterns in diverse software phases and products. Today, we have thousands of patterns (Rising, 2000) for additional topics such as software reuse (Long, 2001), agile software projects (Andrea et al., 2002) or pedagogies (http://www.pedagogicalpatterns.org/) (Abreu, 1997; Fincher & Utting, 2002). Many other patterns are stored in pattern repositories such as the Portland pattern repository (PPR, 2005) or the hillside pattern library (HPL, 2005) and are continuously expanded over conferences such as PLOP (Pattern Languages of Programming; see http://hillside.net/conferences/).

The concept of patterns is used to describe the experience and knowledge that was acquired during projects and have been proven beneficial.

Contrary to (design) patterns, anti-patterns (Brown et al., 1998) are descriptions of problems that commonly occur in software products, processes and projects. Similar to patterns these anti-patterns are described semi-formal based on different templates (Brown et al., 1998) that consist of informal textual or graphical descriptions. However, while patterns typically state and emphasize a single solution to multiple problems, anti-patterns typically state and emphasize a single problem that has potentially multiple solutions. In the literature they are defined as follows:

- *"An Antipattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences"* (Brown et al., 1998)
- *"Antipatterns identify common mistakes"* and *"An Antipattern is defined as a 'commonly occuring solution to a problem that generates decidedly negative consequences'"* (Brown et al., 1999)
- *"An 'antipattern' is similar to a pattern except that it is an obvious but wrong solution to a problem."* (Long, 2001)

- *"… antipatterns describe solutions that have more negative consequences than positive benefits." (Laplante & Neill, 2006)*
- *"An antipattern is a repeated application of code or design that leads to a bad outcome"* (Dudney et al., 2002)
- *"Anti-patterns, also called pitfalls, are classes of commonly-reinvented bad solutions to problems. They are studied as a category so they can be avoided in the future, and so instances of them may be recognized when investigating non-working systems."* (Wikipedia, http://en.wikipedia.org/wiki/Antipattern)
- *"An AntiPattern is a pattern that tells how to go from a problem to a bad solution."* (WikiWikiWeb, http://c2.com/cgi/wiki?AntiPattern)

In summary antipatterns are "bad", "negative", or "worst practices" that describe one problem with potentially many solutions and patterns are "good", "positive", or "best practices" that describe one solution with potentially many problems.

In the following sections we will list most of these anti-patterns that were found in the literature survey. The first large collection of anti-patterns were collected by William J. Brown, Raphael C. Malveau, Hays W. "Skip McCormick III, and Thomas J. Mowbray (Brown et al., 1998).

Table 7. *Antipatterns by (Brown et al., 1998)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
The Blob (God Class)	Structural	Classes, Associations	Maintainability, Portability	Classes with too many functionality and associations to other classes.	●	○	⊙	●
Lava Flow	Structural, Control	Classes, Statements, Associations	Maintainability, Portability	Old or dead code of deprecated or speculative features.	●	○	⊙	●
Functional Decomposition	Structural	Classes, Associations	Maintainability, Portability	Non-OO design is coded in OO language – e.g., by using only one method in a class.	⊙	○	⊙	⊙
Poltergeists	Control, Dynamic	Classes, Statements, Associations	Maintainability, Portability	Classes have very limited roles and life cycles – often starting processes for other objects.	⊙	○	⊙	●
Spaghetti code	Structural, Control	Classes, Methods, Calls	Maintainability, Reliability	Classes call many other classes and the coupling between classes is high. The control flow is jumping through too many classes without clear boundaries.	●	○	⊙	●
Cut & Paste Programming	Semantic, Control	Methods, Statements	Maintainability, Reliability	Code reuse by copying source statements.	●	○	●	○
Stovepipe system	Structural, Control	Packages, Classes, Statements	Maintainability	Many different solutions and absence of abstractions (e.g., large packages, no layers, etc.)	●	○	○	⊙

Furthermore, many anti-patterns were described for J2EE, EJB, or Java. As these anti-patterns are platform-specific only an excerpt of the 52 documented J2EE anti-patterns in (Dudney et al., 2002), the XX EJB anti-patterns in (Tate et al., 2003), or the XX Java and J2EE anti-patterns in (Tate, 2002) are given in the following tables:

Table 8. *Antipatterns by (Dudney et al., 2002)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Localizing Data	Structure	Classes, Statements	Portability	Data is stored and handled in only one location by one element.	⊙	⊙	○	⊙
Multiservice	Structure	Classes, Methods, Interfaces	Maintainability	A service (e.g., class or component) with a large number of public interfaces (resp. responsibilities)	●	⊙	○	●
Tiny Service	Structure	Classes, Methods, Interfaces	Maintainability, Efficiency	A service that only implements a subset of the necessary functionality – resulting in the need to use multiple services for one task.	●	⊙	⊙	●
Too much Code	Control	Classes, Methods	Maintainability, Portability	Too much code ended up in the JSP (or GUI representation).	⊙	⊙	○	●
Sessions A-Plenty	Control	Classes, Methods	Maintainability, Portability	Using sessions for problems that don't need them	⊙	⊙	●	●
Bloated Session	Structure	Classes, Methods, Interfaces	Maintainability	A large Session Bean that implements too many different abstractions.	●	⊙	○	●
Large Transaction	Structure	Methods	Maintainability, Efficiency	A transactional session method that implements a long, complicated process and involves a lot of resources.	●	⊙	●	●
Transparent Façade	Structure, Semantic	Classes, Methods, Interfaces	Maintainability	A façade that directly matches the underlying component – not a coarser-grained interaction.	●	⊙	○	●

Table 9. *Java Antipatterns by (Tate, 2002)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Too many web page items	Structure, Control	JSP, HTML page	Efficiency	Loading too many large items such as graphics.	⊙	⊙	○	⊙
Excessive Layering	Structure	Classes, Inheritance, Layers	Maintainability	Far too many layers of abstraction – e.g., of services or inheritance.	●	⊙	○	⊙
Magic Servlet	Structure	Classes, Methods, Interfaces	Maintainability	A servlet that does all or most of the work itself.	●	⊙	○	●
Monolithic JSP	Structure	Classes, Calls, Statements	Maintainability	A JSP that shows the absence of model-view-controller separation.	⊙	⊙	○	⊙
The Cachless Cow	Control, Dynamic	Statements	Efficiency	Content is very often reloaded without using a cache	⊙	⊙	●	⊙
Lapsed Listeners Leak	Control	Statements	Efficiency, Reliability	An event listener is registered without being removed.	●	⊙	●	●
The Leak Collection	Control	Statements	Efficiency	A collection keeps references to objects that will not be used anymore, until the collection is destroyed late in the lifecycle.	●	⊙	●	●
Connection Thrashing	Control	Statements	Efficiency	Connections to databases are continuously created and destroyed.	●	●	●	⊙

Table 10. *EJB Antipatterns by (Tate et al., 2003)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Local & Remote Interfaces	Structure	Classes, Methods, Interfaces	Maintainability	A class that supports both local and remote interfaces.	●	⭘	○	●
Swallowing Exceptions	Structure, Semantic	Classes, Methods, Exceptions	Maintainability, Reliability	Exceptions are not handled but only logged	●	⭘	⭘	●
Narrow Servlet Bridges	Structure	Servlets, Bridges	Maintainability	Too many Bridges for the servlets	⭘	⭘	○	●
Fat Message	Structure, Control	Classes, Statements	Efficiency	The same message type is used for all situations.	●	⭘	⭘	●
Skinny Message	Structure, Control	Classes, Statements	Efficiency	Messages that don't contain enough information and require the reload of additional information.	●	⭘	⭘	●
Monolithic Consumer	Structure, Control	Classes, Statements	Maintainability	Inlining business logic in classes that consumes a message.	⭘	⭘	⭘	●
Hot Potato	Control, Dynamic	Classes, Statements	Efficiency	A message is tossed back and forth – sometimes because it was not acknowledged.	●	⭘	⭘	●
Face Off	Structure	Beans, Calls	Maintainability, Reliability	A client is directly accessing entity beans.	⭘	○	●	●

Hallal et al. have catalogued 38 anti-patterns that relate to multithreading, concurrency, and synchronization in Java. As they have not provided an extensive description of these anti-patterns we list only a subset described in their paper:

Table 11. *Multithread Antipatterns by (Hallal et al., 2004)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Synchronized method call in cycle of lock graph.	Control	Classes, Methods, Statements	Efficiency, Reliability	Synchronized methods call each other (in a loop).	●	○	●	⭘
Unsynchronized spin-wait.	Control	Classes, Methods, Statements	Efficiency, Reliability	An unsynchronized loop, whose exit condition is controlled by another thread - resulting in the exhaustive use of resources (CPU) and thread stalls.	●	○	●	●
Non synchronized run() method.	Control	Classes, Methods, Statements	Reliability	Different threads are started for an unsynchronized object that implements the Runnable interface.	●	○	●	⭘
Internal call of a method.	Control	Classes, Methods, Statements	Efficiency, Reliability	One thread gets the monitor (lock) several times in a nested way.	●	○	●	⭘
wait() is not in loop.	Control	Statements	Reliability	wait() is used without a loop – but the condition might already have changed.	●	○	●	●
Double call of the start() method of a thread.	Control	Classes, Methods, Statements	Efficiency, Reliability	The start() method call is used more than once for the same thread.	●	○	●	●

Table 12. *Performance Antipatterns by (Parsons & Murphy, 2004a, 2004b)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Too Many Remote Calls	Structure, Control	EJB Beans	Efficiency	Loading too many large items such as graphics or data from remote places (e.g., using getter methods).	○	○	○	○
Aggressive Loading of Entities	Structure, Control	EJB Beans	Efficiency	The loading of an instance of a single entity bean may result in the loading of numerous entity beans from the database, producing a large entity bean graph.	○	○	○	●

Smith and Williams (Smith & Williams, 2001, 2002, 2003) describe and list several performance antipatterns. However, as some of them are abstract and not applicable on the architecture and design level only an excerpt is listed in Table 13.

Table 13. *Performance Antipatterns by (Smith & Williams, 2001, 2002, 2003)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Empty Semi Trucks	Structure, Control	Classes, Statements	Efficiency, Reliability	Occurs when an excessive number of requests is required to perform a task.	○	○	○	●
Roundtripping	Structure, Control	Classes, Statements	Efficiency, Reliability	Many fields in a user interface must be retrieved from a remote system.	○	○	○	●
Sisyphus Database Retrieval	Structure, Control	Classes, Statements	Efficiency, Reliability	Special case of The Ramp. Occurs when performing repeated queries that need only a subset of the results.	○	○	○	●
More is Less	Control, Dynamic	Classes, Statements	Efficiency, Reliability	Too many processes relative to available resources.	◯	○	○	●
“god” Class	Structural	Classes, Attributes, Associations	Efficiency	Occurs when a single class either 1) performs all of the work of an application or 2) holds all of the application’s data. Either manifestation results in excessive message traffic that can degrade performance.	●	○	○	●
Excessive Dynamic Allocation	Control, Dynamic	Classes, Statements	Efficiency	Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance.	○	○	○	●
Circuitous Treasure Hunt	Control, Dynamic	Classes, Statements	Efficiency	Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each “look,” performance will suffer.	○	○	○	●

Finally, larger collections such as Reuse Antipatterns by (Long, 2001) or Managerial Antipatterns by (Laplante & Neill, 2006) are too general to apply to the architecture or design level.

4.6 Bug Patterns

The concept "bug pattern" was coined by Eric Allen in the Book "Bug Patterns in Java" (Allen, 2002). These patterns represent problematic parts of the software system that seem wrong, complicated, or cumbersome to an experienced developer. In general, bug patterns are problems that are associated with one or more specific refactorings (i.e., concrete treatments) that might be applied to remove the patterns. In the literature they are defined as follows:

- *"[bug patterns are] recurring relationships between signaled errors and underlying bugs in a program"* (Allen, 2002)
- *"A bug pattern is an abstraction of a recurring bug. In other words, a bug pattern is a literary form that describes a commonly occurring error in the implementation of the software design." (Farchi et al., 2003)*
- *"Bug patterns are code idioms that are often errors." (D. H. Hovemeyer, 2005)*

The concept of bug patterns is used to describe the experience and knowledge that was acquired by experts and have been proven beneficial.

Beside the bug patterns on the code or design levels many other problems were described using this metaphor. Today, we have bug patterns on different abstraction layers, for development phases, or technologies such as concurrent bug patterns (Farchi et al., 2003), multi-threaded systems (Copty & Shmuel, 2005), performance bug patterns (Galvans, 2006), or bug patterns in general java systems (D. H. Hovemeyer, 2005).

In the following sections we will list most of these bug patterns that were found in the literature survey. The first large collection of bug patterns were collected by Eric Allen (Allen, 2002).

Table 14. *Bug Patterns by (Allen, 2002)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
The Rogue Tile	Semantic	Methods, Statements	Maintainability, Reliability	Bug seems to be fixed, but copy and paste spread it all over the sources -- Use type system inheritance, not the copy and paste derivate.	●	○	●	○
The Dangling Composite	Control	Statements	Reliability	Code that uses a recursively defined data type is signaling a NullPointerException.	⭘	○	●	●
The Null Flag	Control	Statements	Reliability	A code block that uses null pointers as flags for exceptional conditions signals a NullPointerException.	⭘	○	●	●
The Double Descent	Control	Statements	Reliability	A ClassCastException is thrown during recursion -- make only one recurrent step at a time, check your invariants.	●	○	●	●
The Liar View	Control	Statements	Reliability	A GUI program passes a suite of tests, but then exhibits behaviour that should've been ruled out by those tests.	○	○	⭘	○
Saboteur Data	Control	Statements	Reliability	Input data in an invalid format crashes your application - Always parse input data, e.g. with regular expressions or with a full featured parser generator, never ever specify the user behavior.	⭘	○	⭘	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
The Broken Dispatch	Control	Methods, Statements	Maintainability, Reliability	Overloading a method breaks some test cases because the wrong implementation will be called.	●	○	●	●
The Impostor Type	Control	Classes, Attributes, Statements	Maintainability, Reliability	Using special fields inside classes to distinguish conceptually distinct subtypes.	○	○	○	●
The Split Cleaner	Control	Classes, Statements	Maintainability, Reliability	Not all resources are cleaned (especially when an exception is thrown) -- use try ... finally ...	●	○	○	●
The Fictitious Implementation	Control	Classes, Methods, Statements	Maintainability, Reliability	A certain implementation of an interface breaks some invariants.	○	○	○	○
The Orphaned Thread	Control	Classes, Attributes, Statements	Maintainability, Reliability	A multithreaded program locks up with or without printing a stack trace to standard error.	○	○	●	○
The Run-On Initialization	Control	Classes, Attributes, Methods	Maintainability, Reliability	Not all fields of a class are initialized properly -- initialize all fields in the constructor.	●	○	●	●

Table 15. *Bug Patterns by (Farchi et al., 2003)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Nonatomic Operations Assumed to Be Atomic	Control	Statements	Reliability	An operation that "looks" like one operation in one programmer model (e.g., the source code level) but actually consists of several unprotected operations at the lower abstraction levels (e.g., bytecode).	○	○	○	●
Two-Stage Access	Control	Statements	Reliability	A sequence of operations needs to be protected but the programmer wrongly assumes that separately protecting each operation is enough.	○	○	○	●
Wrong Lock or No Lock	Control	Statements	Reliability	A code segment is protected by a lock but other threads do not obtain the same lock instance when executing.	●	○	●	●
Double-checked Locking	Control	Methods, Statements	Reliability	When an object is initialized, the thread local copy of the object's field is initialized but not all object fields are necessarily written to the heap. This might cause the object to be partially initialized while its reference is not null.	○	○	○	●
The sleep()	Control	Statements	Reliability	It is assumed that a child thread should be faster than the parent thread and an "appropriate" sleep() is added to the parent thread. However, the parent thread may still be quicker in some environment.	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Losing a Notify	Control	Statements	Reliability	A notify() is executed before its corresponding wait(), the notify() has no effect and is lost.	○	○	○	●
A "Blocking" Critical Section	Control	Statements	Reliability	A thread is assumed to eventually return control but it never does. This situation may occur in a critical section protocol.	○	○	○	●
The Orphaned Thread	Control	Statements	Reliability	A single, master thread drives the actions of the other threads via messages, often by placing them on a queue, that are then processed by the other threads. If the master thread terminates abnormally, the remaining threads may continue to wait on more input to the queue and causing the system to hang.	○	○	○	○

Table 16. *Bug Patterns by (D. Hovemeyer & Pugh, 2004)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Cloneable Not Implemented Correctly	Control	Statements	Reliability	A class implements the Cloneable interface and does not call super.clone()	○	○	○	●
Double Checked Locking	Control	Statements	Reliability, Maintainability	Usage of the double checked locking pattern that doesn't work	○	○	○	●
Dropped Exception	Control	Statements	Reliability	A try-catch block where the catch block is empty and the exception is slightly discarded.	●	○	●	●
Suspicious Equals Comparison	Control, Structure	Statements, Inheritance, Methods	Reliability, Maintainability	Two objects of types known to be incomparable are compared using the equals() method.	○	○	●	●
Bad Covariant Definition of Equals	Structure	Statements, Inheritance, Methods	Reliability, Maintainability	A covariant version of equals() does not override the version in the Object class, which may lead to unexpected behavior at runtime	○	○	○	●
Equal Objects Must Have Equal Hashcodes	Structure	Statements, Inheritance, Methods	Reliability, Maintainability	A class overrides equals() but not hashCode().	○	○	○	●
Inconsistent Synchronization	Control	Statements, Attributes	Reliability	Access is allowed to mutable fields without synchronization - fields which are sometimes accessed with the lock held and sometimes without are candidate instances of this bug pattern.	●	○	●	○
Static Field Modifiable By Untrusted Code	Control	Statements, Attributes	Reliability	Untrusted code is allowed to modify static fields, thereby modifying the behavior of the library for all users.	○	○	○	●
Null Pointer Dereference	Control	Statements	Reliability	A null value might be dereferenced	●	○	●	●
Redundant Comparison to Null	Control	Statements	Reliability	Comparisons in which the outcome is fixed because either both compared	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
				values are null, or one value is null and the other non-null.				
Non-Short-Circuit Boolean Operator	Control	Statements	Reliability, Maintainability	Use of a non-short-circuit boolean operator where they intended to use a short-circuiting boolean operator.	⭘	○	●	●
Open Stream	Control	Statements	Reliability	A program opens an input or output stream, without closing it.	●	⭘	●	⭘
Read Return Should Be Checked	Control	Statements	Reliability	It is incorrectly assume that read() methods always return the requested number of bytes.	⭘	○	●	●
Return Value Should Be Checked	Control	Statements	Reliability	The return value of a method call on an immutable object is ignored.	●	○	●	●
Non-serializable Serializable Class	Structure, Control	Classes, Inheritance, Attributes	Reliability	Classes that implement the Serializable interface but which cannot be serialized – e.g., due to the fact that the superclass of the class is not serializable	●	○	○	●
Uninitialized Read In Constructor	Control	Statements	Reliability	An unitialized field is read before it is written (in a constructor).	●	○	●	●
Unconditional Wait	Control	Statements	Reliability	Code where a monitor wait is performed unconditionally upon entry to a synchronized block – i.e., a notification performed by another thread could be missed.	●	○	●	●
Wait Not In Loop	Control	Statements	Reliability	A lock is not rechecked - there is a window between the time that the waiting thread is woken and when it reacquires the lock, during which another thread could cause the condition to become false again.	●	○	●	●

4.7 Critic Rules

The concept “critic rules” was one of the first concepts used in the diagnosis of problems in software design. It was coined by Jason E. Robbins in his Ph.D. research (Robbins, 1999) and was implemented in the ArgoUML software design environment. These critic rules represent problematic parts of the software system that detects the break of C2 style guidelines. The design environment does not critique the design so much as the objects in the design representation critique themselves. In general, critic rules are potential problems the designer should reflect about with subjectively defined priorities, and that are associated with one or more specific treatments (i.e., “add subclass”). In the literature they are defined as follows:

- *“Critics are active agents that continually check the design for errors or areas needing improvement” (Robbins, 1999)*
- *“[Critic rules] comment on high-level design issues rather than diagram completeness” (Coelho & Murphy, 2007)*
- *“The output of a critic is a* critique*—a statement about some aspect of the model that does not appear to follow good design practice.” (ArgoUML, 2007)*

The following table list several of the critic rules described by Robbins et al. – however, some rules were excluded as they commune comments by colleagues such as “Portability Questionable” or check against stated goals such as “Not enough Reusable Components”.

Table 17. *Design Critic Rules by (Robbins, 1998, 1999; Robbins et al., 1997, 1998a, 1998b; Robbins et al., 1998c; Robbins & Redmiles, 1998, 2000)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Interface Mismatch	Structure	Classes, Calls	Maintainability, Compilability	Component needs certain messages be sent or received.	●	○	⊙	●
Direct Connection	Structure	Classes, Calls	Maintainability, Compilability	Violation of C2 style guideline – no message bus is used to add components after deployment.	●	○	⊙	●
Missing Memory Requirements	Control	Requirement, Statements	Efficiency, Reliability	The memory required to run this component has not been specified.	○	○	⊙	●
Component Choice	Structure	Classes	Maintainability	Other components could fit in place of the existing component.	⊙	○	○	●
Too Much Memory	Control	Requirement, Statements	Efficiency, Reliability	Calculated memory requirements exceed stated goals.	○	○	⊙	●
Too Many Components	Structure	Classes, Calls	Maintainability	There are too many components at the same level of decomposition.	●	○	○	●
Generator Limitation	?	Classes, Calls	Compilability	The code generator cannot make full use of this component.				
Invalid Connection	Structure	Classes, Calls	Maintainability, Compilability	Mandatory message signatures not satisfied by adjacent components in the conceptual architecture	●	○	⊙	●

After the dissertation of Jason E. Robbins the critiques in ArgoUML were advanced and the collection of critics was extended.

Table 18. *Additional Critics in ArgoUML (ArgoUML, 2007)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
15.3.1. Wrap DataType	Structure	Classes	Maintainability	Wrong usage of DataTypes within UML 1.4.	●	○	○	●
15.3.2. Reduce Classes in diagram	Structure	Classes, Diagrams	Maintainability	Too many classes on a diagram.	●	○	○	●
15.3.3. Clean Up Diagram	Structure	Model elements, Diagrams	Maintainability	Model elements are overlapping.	●	○	○	●
15.4.1. Resolve Association Name Conflict	Structure	Associations	Compilability	Two associations in the same namespace have the same name	●	○	○	○
15.4.2. Revise Attribute Names to Avoid Conflict	Structure	Attributes	Compilability	Two attributes of a class have the same name	●	○	○	⊙
15.4.3. Change Names or Signatures in a model element	Structure	Attributes	Compilability	Two methods have the same signature	●	○	○	⊙
15.4.4. Duplicate	Structure	Associations	Compilability	The specified association has two	●	○	○	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
End (Role) Names for an Association				(or more) ends (roles) with the same name. One of the well-formedness rules in UML 1.4 for associations, is that all end (role) names must be unique.				
15.4.5. Role name conflicts with member	Structure	Associations	Compilability, Maintainability	A suggestions that good design avoids role names for associations that clash with attributes or methods of the source class. Roles may be realized in the code as attributes or operations, causing code generation problems.	●	○	○	●
15.4.6. Choose a Name (Classes and Interfaces)	Structure	Classes, Names	Compilability, Maintainability	The class or interface concerned has been given no name (it will appear in the model as Unnamed)	●	○	○	●
15.4.7. Choose a Unique Name for a model element (Classes and Interfaces)	Semantic	Associations, Names	Compilability, Maintainability	Suggestion that the class or interface specified has the same name as another (in the namespace), which is bad design and will prevent valid code generation.	●	○	○	◘
15.4.8. Choose a Name (Attributes)	Structure	Attributes, Names	Compilability, Maintainability	The attribute concerned has been given no name (it will appear in the model as (Unnamed Attribute)).	●	○	○	●
15.4.9. Choose a Name (Operations)	Structure	Methods, Names	Compilability, Maintainability	The operation concerned has been given no name (it will appear in the model as (Unnamed Operation)).	●	○	○	●
15.4.10. Choose a Name (States)	Structure	States, Names	Compilability, Maintainability	The state concerned has been given no name (it will appear in the model as (Unnamed State)).	●	○	○	●
15.4.11. Choose a Unique Name for a (State related) model element	Semantic	States, Names	Compilability, Maintainability	The state specified has the same name as another (in the current statechart diagram), which will prevent valid code generation.	●	○	○	◘
15.4.12. Revise Name to Avoid Confusion	Semantic	Names	Maintainability	Two names in the same namespace have very similar names (differing only by one character).	●	○	◘	●
15.4.13. Choose a Legal Name	Semantic	Names	Compilability, Conformance, Maintainability	All model element names in ArgoUML must use only letters, digits and underscore characters.	●	○	◘	●
15.4.14. Change a model element to a Non-Reserved Word	Semantic	Names	Compilability, Conformance, Maintainability	Suggestion that this model element's name is the same as a reserved word in UML (or within one character of one), which is not permitted.	●	○	○	●
15.4.15. Choose a Better Operation Name	Semantic	Methods, Names	Conformance, Maintainability	An operation has not followed the naming convention that operation names begin with lower case letters.	●	○	○	●
15.4.16. Choose a Better Attribute Name	Semantic	Attribute, Names	Conformance, Maintainability	An attribute has not followed the naming convention that attribute names begin with lower case letters.	●	○	○	●
15.4.17. Capitalize Class Name	Semantic	Classes, Names	Conformance, Maintainability	A class has not followed the naming convention that classes begin with upper case letters.	●	○	○	●
15.4.18. Revise Package Name	Semantic	Package, Names	Conformance, Maintainability	A package has not followed the naming convention of using lower case letters with periods used to indicated sub-packages.	●	○	○	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
15.5.2. Add Instance Variables to a Class	Structure	Classes, Attributes	Conformance, Maintainability	No instance variables have been specified for the given (non-<<utility>>) class.	●	○	○	●
15.5.3. Add a Constructor to a Class	Structure	Classes, Methods	Reliability, Maintainability	Not all of the classes attributes have initial values and the class has no constructor. Constructors initialize new instances such that their attributes have valid values.	●	○	◎	●
15.5.4. Reduce Attributes on a Class	Structure	Classes, Attributes	Maintainability	The class has too many attributes for a good design, and is at risk of becoming a design bottleneck.	●	○	○	●
15.6.1. Operations in Interfaces must be public	Structure	Classes, Methods	Compilability, Maintainability	Non-public operations in Interfaces	●	○	○	●
15.6.2. Interfaces may only have operations	Structure	Classes, Attributes	Conformance	An interfaces has attributes defined. The UML standard defines interfaces to only have operations.	●	○	○	●
15.6.3. Remove Reference to Specific Subclass	Structure	Classes, Attributes	Conformance	A class should not reference its subclasses directly through attributes, operations or associations.	●	○	○	●
15.7.1. Reduce Transitions on <state>	Structure	States	Maintenance	State is involved in so many transitions it may be a maintenance bottleneck.	●	○	◎	●
15.7.2. Reduce States in machine <machine>	Structure	States	Maintenance	State machine has so many states as to be confusing and should be simplified	●	○	●	●
15.7.3. Add Transitions to <state>	Structure	States	Compilability	State requires both incoming and outgoing transitions	●	○	●	●
15.7.4. Add Incoming Transitions to <model element>	Structure	States	Compilability	State requires incoming transitions	●	○	●	●
15.7.5. Add Outgoing Transitions from <model element>	Structure	States	Compilability	State requires outgoing transitions	●	○	●	●
15.7.6. Remove Extra Initial States	Structure	States	Compilability, Conformance	There is more than one initial state in the state machine or composite state, which is not permitted in UML	●	○	●	●
15.7.7. Place an Initial State	Structure	States	Compilability, Conformance	There is no initial state in the state machine or composite state.	●	○	●	●
15.7.8. Add Trigger or Guard to Transition	Structure	States, Transitions	Compilability, Conformance	A transition is missing either a trigger or guard, one at least of which is required for it to be taken.	●	○	●	●
15.7.9. Change Join Transitions	Structure	States, Transitions	Compilability, Conformance, Maintainability	The join pseudostate has an invalid number of transitions. Normally there should be one outgoing and two or more incoming.	●	○	○	●
15.7.10. Change Fork Transitions	Structure	States, Transitions	Compilability, Conformance, Maintainability	the fork pseudostate has an invalid number of transitions. Normally there should be one incoming and two or more outgoing.	●	○	○	●
15.7.11. Add Choice/Junction Transitions	Structure	States, Transitions	Compilability, Conformance, Maintainability	The branch (choice or junction) pseudostate has an invalid number of transitions. Normally there should be at least one incoming transition	●	○	○	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
				and at least one outgoing transition.				
15.7.12. Add Guard to Transition	Structure	States, Transitions	Compilability, Conformance, Maintainability	Transition requires a guard	●	○	○	●
15.7.14. Make Edge More Visible	Structure	States, Transitions	Maintainability	An edge model element such as an association or abstraction is so short it may be missed.	●	○	○	●
15.7.15. Composite Association End with Multiplicity > 1	Structure	Classes, Attributes	Compilability	An instance may not belong by composition to more than one composite instance.	●	○	○	◘
15.8.1. Consider using Singleton Pattern for <class>	Structure	Classes, Attributes	Maintainability	The class has no non-static attributes nor any associations that are navigable away from instances of this class.	●	○	○	●
15.8.2. Singleton Stereotype Violated in <class>	Structure	Classes, Stereotypes	Conformance, Functionality, Reliability, Maintainability	This class is marked with the «singleton» stereotype, but it does not satisfy the constraints imposed on singletons.	●	○	○	●
15.8.3. Nodes normally have no enclosers	Structure	Deployment	Conformance, Maintainability	Nodes should not be drawn inside other model elements on the deployment diagram	●	○	○	●
15.8.4. NodeInstances normally have no enclosers	Structure	Deployment	Conformance, Maintainability	node instances should not be drawn inside other model elements on the deployment diagram	●	○	○	●
15.8.5. Components normally are inside nodes	Structure	Deployment	Conformance, Maintainability	Components represent the logical entities within physical nodes, and so should be drawn within a node.	●	○	○	●
15.8.6. ComponentInstances normally are inside nodes	Structure	Deployment	Conformance, Maintainability	Components instances represent the logical entities within physical nodes, and so should be drawn within a node	●	○	○	●
15.8.7. Classes normally are inside components	Structure	Deployment	Conformance, Maintainability	Classes, as model elements making up components, should be drawn within components on the deployment diagram	●	○	○	●
15.8.8. Interfaces normally are inside components	Structure	Deployment	Conformance, Maintainability	Interfaces, as model elements making up components, should be drawn within components on the deployment diagram	●	○	○	●
15.8.9. Objects normally are inside components	Structure	Deployment	Conformance, Maintainability	Objects, as instances of model elements making up components, should be drawn within components or component instances on the deployment diagram.	●	○	○	●
15.8.10. LinkEnds have not the same locations	Structure	Deployment	Conformance, Maintainability	A link (e.g. association) connecting objects on a deployment diagram has one end in a component and the other in a component instance (since objects can be in either).	●	○	○	●
15.8.11. Set classifier (Deployment Diagram)	Structure	Deployment	Conformance, Maintainability	An instance (object) without an associated classifier (class, datatype) on a deployment diagram.	●	○	○	●
15.8.12. Missing return-actions	Control	Sequence	Compilability, Conformance	A sequence diagram has a send or call action without a corresponding return action.	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
15.8.13. Missing call(send)-action	Control	Sequence	Compilability	A sequence diagram has a return action, but no preceding call or send action.	●	○	●	●
15.8.14. No Stimuli on these links	Control	Sequence	Compilability, Maintainability	A sequence diagram has a link connecting objects without an associated stimulus (without which the link is meaningless).	●	○	●	●
15.8.15. Set Classifier (Sequence Diagram)	Control	Sequence	Compilability, Maintainability	An object without an associated classifier (class, datatype) on a sequence diagram.	●	○	●	●
15.8.16. Wrong position of these stimuli	Control	Sequence	Compilability, Maintainability	The initiation of send/call-return message exchanges in a sequence diagram does not properly initiate from left to right.	●	○	●	●
15.9.1. Circular Association	Structure	Association	Compilability	An association class has a role that refers back directly to itself, which is not permitted.	●	○	○	●
15.9.2. Make <association> Navigable	Structure	Association	Compilability	The association referred to is not navigable in either direction.	●	○	○	●
15.9.3. Remove Navigation from Interface via <association>	Structure	Association	Compilability	Associations involving an interface can be not be navigable in the direction from the interface.	●	○	○	●
15.9.4. Add Associations to <model element>	Structure	Association	Compilability, Maintainability	The specified model element (actor, use case or class) has no associations connecting it to other model elements.	●	○	○	●
15.9.6. Reduce Associations on <model element>	Structure	Association	Compilability, Maintainability	The given model element (actor, use case, class or interface) has so many associations it may be a maintenance bottleneck.	●	○	○	●
15.11.1. Classifier not in Namespace of its Association	Semantic	Association	Compilability, Conformance	All the classifiers attached to the ends of the association should belong to the same namespace as the association.	●	○	○	●
15.11.2. Add Elements to Package <package>	Structure	Package	Maintainability	The specified package has no content.	●	○	○	●
15.13.2. Class Must be Abstract	Structure	Classes, Methods	Compilability	A class that inherits or defines abstract operations must be marked abstract.	●	○	○	●
15.13.3. Add Operations to <class>	Structure	Classes, Methods	Maintainability	The specified class has no operations defined.	●	○	○	●
15.13.4. Reduce Operations on <model element>	Structure	Classes, Methods	Maintainability	The model element (class or interface) has too many operations	●	○	○	●
15.14.1. Change Multiple Inheritance to interfaces	Structure	Classes, Inheritance	Maintainability	A class has multiple generalizations, which is permitted by UML, but cannot be generated into Java code, because Java does not support multiple inheritance.	◘	○	○	●
15.16.2. Remove <class>'s Circular Inheritance	Structure	Classes, Inheritance	Compilability	A class inherits from itself, through a chain of generalizations, which is not permitted.	●	○	○	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
15.16.4. Remove final keyword or remove subclasses	Structure	Classes, Inheritance	Compilability	A class that is final has specializations, which is not permitted in UML.	●	○	○	●
15.16.5. Illegal Generalization	Structure	Classes, Inheritance	Compilability	A generalization between model elements of different UML metaclasses, which is not permitted.	●	○	○	●
15.16.6. Remove Unneeded Realizes from <class>	Structure	Classes, Inheritance	Maintainability	A realization relationship both directly and indirectly to the same interface (by realization from two interfaces, one of which is a generalization of the other for example).	●	○	○	●
15.16.7. Define Concrete (Sub)Class	Structure	Classes, Inheritance	Maintainability	A class is abstract with no concrete subclasses, and so can never be realized.	●	○	○	●
15.16.8. Define Class to Implement <interface>	Structure	Classes, Inheritance	Maintainability	The interface referred to has no influence on the running system, since it is never implemented by a class.	●	○	○	●
15.17.1. Remove Circular Composition	Structure	Classes, Association	Compilability	A series of composition relationships that form a cycle, which is not permitted.	●	○	○	●
15.17.2. Duplicate Parameter Name	Structure	Methods, Parameters	Compilability	A parameter list to an operation or event has two or more parameters with the same name, which is not permitted.	●	○	◘	●
15.17.3. Two Aggregate Ends (Roles) in Binary Association	Structure	Methods, Parameters	Compilability	Only one end (role) of a binary association can be aggregate or composite.	●	○	○	●
15.17.4. Aggregate End (Role) in 3-way (or More) Association	Structure	Associations	Compilability	Three-way (or more) associations can not have aggregate ends (roles).	●	○	○	●

A recent development of critiques by Coelho and Murphy includes additional critiques that motivate to reflect about the software design. However, several critiques seem very context-specific (e.g., the “Plural Contained Class” rule would fire at every use of a container such as Persons).

Table 19. *Critic Rules by (Coelho & Murphy, 2007)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Class References Subclass	Structure	Classes, Attributes, Calls	Maintainability, Portability	A class references a subclass of itself	●	○	○	●
Superclass Reference	Structure	Classes, Calls	Maintainability, Portability	A class references its superclass, but not through an aggregation.	●	○	○	●
Circular Containment	Structure	Classes, Attributes	Maintainability, Portability	There is a cycle in aggregation or composition relationships	●	○	○	○
Association Cycle	Structure	Classes, Calls	Maintainability, Portability	There is a cycle in association Relationships	●	○	◘	○

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Multiple Paths	Structure	Classes, Attributes, Calls	Maintainability, Portability	There are two navigable paths from one class to another	●	○	◘	◘
Duplicated Superclass Reference	Structure	Classes, Calls	Maintainability	A class has an association that is already defined by its superclass	●	○	○	●
Generalizable Aggregation	Structure	Classes, Attributes, Inheritance	Maintainability	A class aggregates two classes that share a superclass	●	○	○	●
Subclass and Superclass Aggregation	Structure	Classes, Attributes, Inheritance	Maintainability	A class aggregates a class and a subclass or superclass of that class	●	○	○	●
Unnecessary Realization	Structure	Classes, Attributes, Inheritance	Maintainability	A class realizes two interfaces that extend each other	●	○	○	●
Plural Contained Class	Semantic	Class, Names	Maintainability	The target of an aggregation or composition has a plural name (which wrongly suggests that it is the container)	●	○	○	●
Method in Attribute Compartment	Semantic	Classes, Attributes, Names	Functionality	There are parentheses in the name of an attribute, which may occur if the user creates a method in the wrong compartment	●	○	○	●
Get or Set Attribute Prefix	Semantic	Classes, Attributes, Names	Functionality, Maintainability	An attribute name begins with get or set, which suggests the user may have put a method name in the attribute compartment	●	○	○	●
Duplicate Class Name	Semantic	Classes, Names	Functionality, Maintainability	Two classes or interfaces in the design have the same name	●	○	○	○
Highly Coupled Design	Structure	Classes, Attributes, Calls	Maintainability	The number of associations, compositions, and aggregations has exceeded some constant multiple of the number of classes	●	○	○	●
Class Has Too Many Associations	Structure	Classes, Calls	Maintainability, Portability	A class has more than some constant number of associations to other classes	●	○	○	●
Duplicated Members	Structure	Classes, Attributes, Methods	Maintainability	Two non-related classes have at least three members in common	●	○	○	○
Missing Attribute	Structure	Classes, Attributes, Methods	Functionality	A getter and setter are defined, but no matching attribute exists	●	○	○	●
Unnecessary Accessors	Structure	Classes, Attributes, Methods	Functionality, Maintainability	A class has more than some constant number of pairs of getter and setters, which may be an unnecessary source of clutter	●	○	○	●
Redeclared Superclass Attribute	Structure	Classes, Attributes	Functionality, Maintainability, Reliability	A subclass redeclares an attribute defined by its superclass.	●	○	○	●

4.8 Defect Patterns

The concept “Defect Pattern” was used by Taiga Nakamura in his “HPC Bug Base“ portal to describe classes of recurring problems (as well as individual defects) in HPC. These defect patterns represent problematic parts of distributed software systems (especially for high per-

formance computing) that seem wrong, complicated, or cumbersome to an experienced developer. In the literature they are defined as follows:

- *"[defect patterns are] functional bugs, performance bottlenecks, portability problems, bad practices, etc. in HPC "(Nakamura, 2007)*

The concept of defect patterns is used to describe the experience and knowledge that was acquired by experts or in empirical evaluations and we will list most of these defect patterns that were found in the literature survey.

Table 20. *Defect Patterns individuals by (Nakamura, 2007)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Missing Wait	Control	Statements	Efficiency	Send and receive without a wait between.	●	○	●	⭘
Bottleneck in Message Scheduling	Control	Statements	Efficiency	In the programming with explicit message passing, inappropriate message scheduling can cause performance bottleneck.	●	○	●	⭘
Bottleneck with File I/O	Control	Statements	Efficiency	When multiple processes access the file or filesystem at the same time, they can cause a performance problem.	⭘	⭘	●	⭘
Calling omp get num threads in a Serial Section	Control	Statements	Efficiency, Reliability	omp_get_num_threads() returns the number of threads currently executing the parallel section where it is called. If it is called in a serial section, the return value is always 1.	⭘	○	●	⭘
Calling upc free from Multiple Threads	Control	Statements	Efficiency, Reliability	Only one thread may call upc_free for each allocation. This is confusing especially if the object was allocated with upc_global_alloc, which is a collective operation.	⭘	○	●	⭘
Corrupted File Output	Control	Statements	Efficiency	HPC applications often need to write to a file to store intermediate and/or final results. If the data is written to the same file by multiple processes/threads at once, the file content can get corrupted.	⭘	⭘	●	⭘
Dependency on the Number of Processes	Control	Statements	Portability	An implementation that only works with specific number of processes is not portable.	●	○	●	●
Excessive Use of Collective Communication	Control	Statements	Efficiency, Portability	Collective communication is commonly used in parallel programming, but there is a concern that it does not scale up well when the number of processes (or threads) increases.	⭘	○	●	⭘
Hidden Serialization in Library Functions	Control	Statements	Efficiency	Library function implementation sometimes contains internal serialization. In a parallel context, it can cause a performance bottleneck.	⭘	○	●	⭘
Inadequate Communication Pattern	Control	Statements	Efficiency	An inadequate communication pattern (e.g., star pattern) can lead to a performance overhead.	⭘	○	●	⭘

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Memory Allocation Failure Due to Inappropriate Compiler Flags	Control	Statements	Efficiency	Full memory access is sometime only available if you compile with the "right" flags. There are also no warning signs that you are running out of memory until it happens.	○	○	○	●
Message Type Mismatch	Control	Statements	Reliability	If the data type and the number of elements do not match between sender and receiver, it can cause a failure at runtime.	●	○	⊙	⊙
Missing MPI Finalize	Control	Statements	Conformance	The MPI specification says that all processes must call MPI_Finalize before exiting.	⊙	○	●	●
Missing upc barrier before exit	Control	Statements	Reliability	In a UPC program, upc_barrier should be called before exit to prevent an issue with some threads exiting before others finish using the data.	⊙	○	●	●
Overlapped Memory Areas	Control	Statements	Efficiency, Reliability	Some MPI functions take a send buffer and a recv buffer. The memory area for these buffers may not overlap.	⊙	○	●	●
Fragmented Messages	Control	Statements	Efficiency	Messages between processes should be aggregated into the chunks of sufficient size to avoid the overhead of connection handshaking and message headers.	●	○	●	●
Passing NULL to MPI Init	Control	Statements	Reliability	In the C version MPI_Init takes two parameters - in MPI 1.1, calling MPI_Init with NULL parameters in come implementation can fail.	⊙	○	●	●
Potential Deadlock	Control	Statements	Efficiency, Reliability	MPI_Send() and MPI_Recv() are the source of potential deadlocks.	⊙	○	⊙	⊙
Upc memget and upc memput from/to Multiple Threads	Control	Statements	Reliability	Trying to copy data from/to multiple different threads results in an error.	⊙	○	●	●
Using the Same Randomization Seed in All Processes	Control	Statements	Reliability, Functionality	Some pseudo-random number libraries require an explicit initilization with a 'seed' which determines the actual sequence to be generated.	●	○	●	●

4.9 Defects, Bugs & Errors (Design)

The concepts (design-oriented) “defects”, “bugs” or “errors” are typically used as a demotic term. However, several authors use the term to describe recurring and named problems. In the literature they are defined as follows:

- *“[A Design defect] is an imperfection in the software engineering work product that requires rectification” (Younessi, 2002)*
- *“Software defects are requirement, design, and implementation errors in a software system” (Telles & Hsieh, 2001)*
- *“Bugs are behaviour of the system that the software development team (developers, testers, and project managers) and customers have agreed are undesirable” (Telles & Hsieh, 2001)*

- *"A consistency defect is a mismatch between overlapping diagrams." (Christian F. J. Lange, 2006)*

Beside the defects on the code levels many other problems were described using the defect metaphor. For example, (Moha & Guéhéneuc, 2005) used the term Software architectural defects are a concept used as an umbrella term similar to design flaws.

The concept of "bugs" was used by Telles and Hsieh in the Book "The Science of Debugging" (Telles & Hsieh, 2001) to describe concrete locations where debugging should take place. They constructed a classification of bugs that starts with abstract classes such as requirement, design, implementation, process, build, deployment, documentation, and future planning bugs. Thereafter, they describe several more specific bug classes that describe recurring problems and mostly are on the level of other quality defects. These bugs mostly represent functional problems of the software system that are associated with one or more specific approach of debugging.

Table 21. *Defect Bug classes by (Telles & Hsieh, 2001)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Memory or resource leaks	Control	Classes, Data	Efficiency	Memory is allocated and used but never freed	⊙	⊙	⊙	●
Logic Errors	Semantic	Methods, Statements	Functionality, Reliability	Code is syntactically correct but does not do what is expected.	●	○	●	●
Coding Errors	Semantic	Methods, Attributes, Parameters	Maintainability, Reliability	A simple problem in writing the code.	●	○	⊙	⊙
Memory Overruns	Control	Classes, Data	Efficiency	Using memory that does not belong to the system.	○	○	⊙	⊙
Loop Errors	Control	Methods, Statements	Functionality, Reliability	Problems with loops such as infinite, nonprocessed, off-by-one, and improperly loops.	●	○	●	●
Conditional errors	Control	Methods, Statements	Functionality, Reliability	Poorly written conditional logic due to misunderstanding or mis-placement of nested conditionals.	●	○	●	●
Pointer Errors	Control	Methods, Statements	Functionality, Reliability	Pointers get messed up and do not point to where they should.	●	○	●	●
Allocation / Deallocation Errors	Control	Methods, Statements	Functionality, Reliability	The order of allocation and deallocation is incorrect.	●	○	●	●
Multithreaded Errors	Control	Methods, Statements	Functionality, Reliability	Two threads try to access or modify the same memory address.	○	⊙	⊙	○
Timing Errors	Control	Timing, Sequence, Statements	Functionality, Reliability	Events were designed to occur at a certain time but doesn't.	⊙	○	⊙	○
Distributed Application Errors	Control	Deployment, Interaction, Statements	Functionality, Reliability	An error in the interface between any two applications in a distributed system.	⊙	⊙	⊙	○
Storage Errors	Data	Data, Statements	Functionality, Reliability	A persistent storage device encounters an error and is unable to proceed.	⊙	●	○	⊙
Integration Errors	Control	Calls	Functionality, Reliability	The integration of two subsystems causes an error.	⊙	⊙	⊙	○
Conversion Errors	Control	Calls	Functionality, Reliability, Maintainability	Data formats are used in a wrong way (esp. between components)	●	●	⊙	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Hard-coded Lengths/Sizes	Control	Statements	Maintainability, Reliability	Constants that appear multiple times in the system	●	●	●	○
Versioning Bugs	Historic	Versions	Functionality, Reliability, Maintainability	Change of functionality or data formats between versions	⭘	⭘	●	○
Inappropriate Reuse Bugs	Control, Data	Statements, Data	Functionality, Reliability, Maintainability	Inappropriate reuse of code or components.	⭘	⭘	●	○
Boolean Bugs	Control	Statements	Functionality, Reliability, Maintainability	Misunderstandings about what a Boolean expression (e.g., true and false) means in the code.	●	○	●	●

The concept of "errors" was used by many authors in to describe problems in software systems. Livshits and Lam use it to describe security errors (resp. security vulnerabilities) (Livshits & Lam, 2005). These errors represent recurring problems of a software system.

Table 22. *Security Errors by (Livshits & Lam, 2005)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
SQL injection	Control	Statements	Functionality (Security)	Pass input containing SQL commands to a database server for execution	●	●	⭘	●
Cross-site scripting	Control	Statements	Functionality (Security)	Exploit applications that output unchecked input verbatim to trick the user into executing malicious scripts	⭘	○	⭘	●
HTTP response splitting	Control	HTTP response	Functionality (Security)	Exploit applications that output input verbatim to perform Web page defacements or Web cache poisoning attacks	⭘	○	⭘	●
Path traversal	Control	URL input parameters	Functionality (Security)	Exploit unchecked user input to control which files are accessed on the server	⭘	○	⭘	●
Command injection	Control	Statements	Functionality (Security)	Exploit user input to execute shell commands.	●	○	●	●
Parameter tampering	Control	Statements	Functionality (Security)	Pass specially crafted malicious values in fields of HTML forms	●	●	⭘	●
URL manipulation	Control	URL input parameters	Functionality (Security)	Use specially crafted parameters to be submitted to the Web application as part of the URL.	●	●	⭘	●
Hidden field manipulation	Control	URL input parameters	Functionality (Security)	Set hidden fields of HTML forms in Web pages to malicious values	●	●	⭘	●
HTTP header tampering	Control	URL input parameters	Functionality (Security)	Manipulate parts of HTTP requests sent to the application	●	●	⭘	●
Cookie poisoning	Control	Statements, Cookie access	Functionality (Security)	Place malicious data in cookies, small files sent to Web-based applications	●	●	⭘	●

The concept of "design defects" was used by Houman Younessi in the book "Object-oriented Defect Management" (Younessi, 2002). These defects represent problematic parts of the software system that are associated with one or more specific UML diagram (e.g., statement

diagrams). Some of them are very function-oriented – i.e., if this defect does exist the model shouldn't compile.

Table 23. *Design Defects by (Younessi, 2002), Chapter 6*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Null Diagram	State	State diagram	Maintainability	Nothing happens between the start and end state.	●	○	⊙	⊙
Trap state	State	State diagram	Functionality, Maintainability, Reliability	A state that can be entered but never exited (i.e., no path to the stop state).	●	○	⊙	●
Tightly Circular	State	State diagram	Maintainability	A tightly circular or reflexive form due to a limited number of states (i.e., all states build a small circle).	●	○	⊙	⊙
Disjoint States	State	State diagram	Maintainability	Independent state paths/streams in one diagram	●	○	⊙	⊙
Deadlock	State	State diagram	Maintainability	The next transition from a state cannot logically take place.	●	○	⊙	●
Conflict	State	State diagram	Maintainability	The transition from a guard or sync point cannot logically take place	●	○	⊙	●
God state	State	State diagram	Maintainability	One event causes many resulting events (e.g., a very small fan-in to fan-out ratio)	●	○	⊙	●
Hub state	State	State diagram	Maintainability	Many independent fan-in events and many independent fan-out events (e.g., a fan-in fan-out ratio near to 1)	●	○	⊙	●
Minion state	State	State diagram	Maintainability	Many events causes only one or very few resulting events (e.g., a very large fan-in to fan-out ratio)	●	○	⊙	●

Furthermore, Younessi lists many design defects in his inspection checklists (see appendix C od (Younessi, 2002)).

Table 24. *Design Defects by (Younessi, 2002), Appendix C*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Global Variables	Structure, Control	Calls	Maintainability	There exists an externally declared variable that is referenced within a function but has not been passed in as a parameter.	●	○	⊙	●
Poor Naming Conventions	Semantic	Names	Maintainability	Identifiers are too long, consist of single characters (except loop indexes), or resemble a keyword.	●	⊙	⊙	●
Redundant Declarations	Structure, Control	Statements	Maintainability	Variables, parameters, or functions that are declared in one class, function, or compound statement but never actually used in that context.	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Input Coupling	Structure, Control	Statements	Maintainability	A variable that is assigned a value via an input, but is not modified or referenced before being passed to a user-defined function as a function call argument.	●	○	●	●
Magic Numbers	Control	Attributes, Statements	Maintainability	There exist numbers other than -1, 0, or 1 in a program statement.	●	●	●	○
Hidden Loops	Control	Statements	Maintainability, Reliability	A guarded variable from a loop guard or a branch guard is modified within a single branch of a guarded statement, is modified within the loop body, or is assigned a value independent of itself within an "if" statement branch (if this variable is not a loop guard variable, it must occur in the guard of the branch).	●	◎	●	●
Uninitialized Variables	Control	Statements	Functionality, Reliability	There exists a variable that has not been explicitly initialized prior to its first use in an expression.	●	○	●	●
Lax Grouping	Control	Statements	Maintainability	Identical subexpressions in each expression of two conditional (if) statements, but there are no statements between the guards of the two (if) statements that modify the variables occurring in the aforementioned subexpressions.	●	○	●	◎
Zero Iteration Defect	Control	Statements	Functionality, Reliability	A variable occurs in a loop body (not in a guard), that: a) is not initialized before the loop, and b) is assigned but not referenced within the loop body, and c) after being assigned, does not appear in an inner loop.	●	○	●	●
Superfluous Variables in Loop: (Does Not Apply to Loop Control Variable)	Control	Statements	Maintainability	Temporary variables in a loop that do not save time in computation or a non-accumulative assignment to a variable in a loop that appears (just once) in the right-hand side of a subsequent assignment statement.	●	○	●	●
Loops That May Make No Progress	Control	Statements	Maintainability	There exist no variables from the loop guard of a loop that are updated within the body, except inside another guarded command.	●	○	●	●
Redundant Loop Computations	Control	Statements	Maintainability	There exists a subexpression that is evaluated within a loop and involves variables that are not changed within the loop (these variables are global to the loop body scope).	●	○	●	●
Loop Guard Too Complex	Control	Statements	Maintainability	There exists a loop statement that contains more than two conditional constructs within its loop guard.	●	◎	●	●
Loop Contains Post-termination Structure	Control	Statements	Maintainability	A loop body that contains a conditional (if) statement, whose block's last statement breaks out (e.g., a break statement).	●	○	●	●
Redundant Conditional Assignment	Control	Statements	Maintainability	An equality guard component for a conditional statement that matches	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
				an assignment statement that it guards, but the variables in the guard are not modified prior to the execution of the matching assignment.				
Self-Assignment	Control	Statements	Maintainability	An assignment statement in which the left-hand side and right-hand side are identical	●	○	●	●
Dispersed Initialization	Control	Statements	Maintainability	A variable that is a control variable of a loop, initialized more than five statements away from where it is employed in the loop, not referenced or modified after the initialization and before the loop.	●	○	●	●
Premature Initialization	Control	Statements	Maintainability, Efficiency	A loop control variable for an inner nested loop that is initialized twice: once before entering the external loop and once before entering or on leaving the inner loop.	●	○	●	●
Redundant Accumulation	Control	Statements	Maintainability	There exist two or more congruent accumulative statements within a loop that are of the form i = i + c1 and j = j + c2, where c1 = c2.	●	○	●	●
Redundant Test on Loop Exit	Control	Statements	Maintainability	An extra guard to test the exit condition of the guarded loop after a loop statement. Between the guard and the loop exit, there exist no statements to change the vaiiables that occur in the guard.	●	○	●	●
Redundant Guard	Control	Statements	Maintainability	A subexpression within a loop of a conditional statement that has previously been established for the given execution path.	●	○	●	●
Readjustment of Loop Variable on Exit	Control	Statements	Maintainability	An expression or statement that readjusts a loop variable on exit from a loop.	●	○	●	●
Redundant Internal Guard	Control	Statements	Maintainability	A guard component that is applied more than once in a loop body without changing its component variables.	●	○	●	⊙
Statement Duplication	Control	Statements	Maintainability	A statement that occurs more than once within a loop body, although between these duplicated statements, the variables they contain are not changed.	●	○	●	○
Duplicate Output	Control	Statements	Maintainability	There exists a variable that is output via an output function and unmodified before being output again by another output function.	●	⊙	●	○
Function Comments	Semantic	Notes / Comments	Maintainability	No comments describing the class, attribute, or function's job, either before or after the functions heading.	●	○	⊙	●
Multiple Exits from a Function	Control	Statements	Maintainability, Reliability	There exists more than one exit statement within a function body.	●	○	●	○
Unassigned Address Parameter	Control	Statements	Maintainability	There exists an address parameter that is not assigned to a value within a function.	●	○	●	○

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Function Side Effects	Control	Statements	Maintainability, Reliability	A function that returns a value; and there exists: An address parameter that may be used to change the contents of a corresponding actual parameter; or an external variable that is changed inside this function.	●	○	●	◎
Amended Nonaddress Parameter	Control	Statements	Maintainability	There exists a nonaddress parameter that is amended inside the body of a function.	●	○	●	●
Redundant Guard Test	Control	Statements	Maintainability	A variable that occurs in two relational expressions joined by the (AND) operator.	●	○	●	●
Indirectly Terminated Loops	Control	Statements	Maintainability, Reliability	A single variable that is used as a guard of an iterative statement, assigned within a guarded statement (selection or iterative statements) within the loop body.	●	○	●	●
Dual-Purpose Variable Usage	Control	Statements	Maintainability, Reliability	There exists a variable that is modified in the body of a loop, then reassigned after the loop.	●	○	●	●
Double Initialization	Control	Statements	Maintainability, Efficiency	A loop variable that is initialized more than once prior to its use in a loop, although the variable is not referenced between the statements in which it is initialized.	●	○	●	○
Subscript Within Bounds	Control	Statements	Maintainability, Reliability	There exists an array the subscripts for which exceed the bounds.	●	○	●	●
Noninteger Subscript	Control	Statements	Maintainability, Reliability	Subscripts of an array should always be integers.	●	○	●	●
Incorrect Initialization	Control	Statements	Maintainability, Reliability	Arrays and strings are usually required to be set to default values.	●	○	●	●
Procedure That Returns a Value as a Parameter	Control	Statements	Maintainability, Reliability, Portability	A procedure that has been specified to return no value but has an address parameter that is assigned within the body (i.e., side-effect).	●	○	●	◎
Operation with No Visible Effect	Control	Statements	Maintainability	An operation that has no effect (i.e., side-effect or return value).	●	○	●	◎
Overloaded Loop Index	Control	Statements	Maintainability, Reliability	There exists an inner loop that changes an outer loop control variable.	●	○	●	◎
Mixed-Mode Computation	Control	Methods, Statements	Maintainability, Reliability	Types do not conform for correct computation.	●	○	●	●
Division by Zero	Control	Statements	Reliability	The denominator of operation has not been guarded against evaluating to zero.	●	○	●	●
Integer Division	Control	Statements	Reliability	Integer division truncates the remainder	●	○	●	●
External Object Attribute Hard-Coded	Control	Attributes, Statements	Maintainability, Portability	The attributes of external objects (e.g., a file) have been explicitly hard-coded.	●	○	●	●
Function Has No Return Value	Control	Methods	Maintainability, Reliability	A (nonvoid) function that contains a return statement with no return value.	●	○	●	●
Unused Input	Control	Statements	Maintainability	A variable that is assigned a value via an input function, not used or referenced until being assigned another	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
				value by another input function.				
Unmodified Output	Control	Statements	Maintainability, Reliability	There exists a variable that is assigned a value from an input function, unmodified before being output again by an output function.	●	○	●	●
Identifiers in Scope Are Character Similar	Semantic	Names	Maintainability, Reliability	Two or more entities (functions, variables, parameters, …) with similar names.	●	○	●	○
Identifiers in Scope Differing in Case	Semantic	Names	Maintainability, Reliability	Two or more entity names (functions, variables, parameters, …) differ only in case.	●	○	●	○
McCabe's Cyclomatic Complexity	Control	Statements	Maintainability	A cyclomatic complexity value of more than 5 indicates that the function is too complex and should be reduced, if possible.	●	○	●	●
Unacceptable Initialization of Global Variables	Control	Statements	Maintainability	A global variable that is initialized in its variable declaration without the use of the appropriate keyword.	●	○	●	●
Local Variables Not Declared within Their Minimal Scope	Control	Statements	Maintainability	Local variables that are declared for a block, not referenced at the top level of that block, but within an inner block (at a lower level).	●	○	●	●
Unintentional Empty Loop	Control	Statements	Maintainability	There exists a loop within an empty body.	●	○	●	●
Inconsistent Use of Delimiters	Control	Statements	Maintainability	There exists a body within a conditional (if-else) statement that is enclosed in delimiters (e.g., a compound statement), whereas the other body counterpart is not a compound statement.	●	○	●	◎
Multiple Breaks in Loop	Control	Statements	Maintainability	There exists more than one break statement within the body of an iterative statement.	●	○	●	●
Redeclaration of Identifiers	Control	Attributes, Variables, Parameters, Statements	Maintainability	The name of an entity (i.e., class variable, parameter, etc.) is redeclared as a local variable in a lower block (e.g., as a local variable)	●	○	●	●
No Break at End of Multiple Branching (case) Statement	Control	Statements	Maintainability, Reliability	The last statement in the body of a multiple branching statement is not a break statement.	●	○	●	●
Default Is Not the Last Label in a Multiple Branch	Control	Statements	Maintainability, Reliability	A multiple branch statement that where the default label does not occur as the last label.	●	○	●	●
Noncompound Multiple Branch Body	Control	Statements	Maintainability, Reliability	There exists a multiple branch statement where its body statement is not a compound statement.	●	○	●	●
Multiple Branch Statement Fall-Through	Control	Statements	Maintainability, Reliability	There exists a top-level case (or default) labeled statement within the body of a multiple branch statement, which is not the first case (or default) labeled statement in the body, and is not preceded by a break statement.	●	○	●	●
Goto Statements Considered Harmful	Control	Statements	Maintainability, Reliability	There exists a goto statement within the body of a function.	●	○	●	●
Allocating Nonavail-	Dynamic	Available	Maintainability,	Memory has been allocated where	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
able Memory		Memory, Statements	Reliability, Efficiency	system memory has been exhausted.				
Memory Is Deallocated Improperly.	Control	Statements	Maintainability, Reliability, Efficiency	Memory has been deallocated without using a removal routine (e.g., destructor), using the proper removal routine, or in an object by another object.	●	○	●	●
The Assignment Operator Returns Unexpected Type	Control	Methods, Statements	Maintainability, Reliability	The type of the object and that of the return type of the assignment operator do not match.	●	○	◘	●
Assignment Operator Attribute Missing	Control	Statements	Maintainability, Reliability	Attributes have been omitted while overloading the assignment operator.	●	○	◘	●
Passing Derived Class Objects by Value	Structure	Classes, Parameters	Maintainability, Reliability	A derived class passed by value is not treated as a base class (it should be)	●	○	○	●
Out-of-Order Initialization of Base Classes	Structure	Classes, Inheritance, Statements	Maintainability, Reliability	In most multiple-inheritance situations, the order of declaration of base classes matters.	●	○	○	●
Inheriting the same feature from more than one class	Structure	Classes, Inheritance	Maintainability, Reliability	The branch from which the feature is to be inherited has not been made explicit.	●	○	○	●
Improper Exception Management	Structure	Methods, Exceptions, Statements	Maintainability, Reliability	An exception propagates beyond the scope, is ignored, is passed from a server to a client improperly, or is wrong. Exception-handling mechanism is missing, incorrect, or falls into an infinite loop.	●	○	◘	●
Improper Inheritance Implementation	Structure	Classes, Inheritance	Maintainability, Reliability	An unnecessary or inappropriate feature has been inherited by a subclass. Subclass violates the invariant of its superclass. Subclass violates the precondition of the superclass. Subclass implements specification or restriction. A feature that is supposed to be implemented in a subclass is missing. Superclass is not initialized. Superclass initialization is incorrect. Visibility rules have been violated.	●	○	○	◘
Improper Assertion	Control	Methods, Statements	Maintainability, Reliability	Precondition not checked at entry. Postcondition not ensured at exit. Class invariant not checked at construction, at entering a precondition, and at exit Modal assertions not checked (for modal classes).	●	○	●	◘
Use of Instance Operators with Expanded Types	Control	Methods, Statements	Maintainability, Reliability	An instance operator has been used with an expanded type (primitive type)	●	○	◘	●
The \ Character Misrepresented	Control	Statements	Maintainability, Reliability	The \ character has not been represented as string \\.	●	○	●	●
Substring Extraction Offsets Used Incorrectly	Control	Statements	Maintainability, Reliability	The substring offsets have been used in a relative fashion, as opposed to two zerobased offsets: one pointing to the start, the other to the character one past the end.	●	○	●	●
Strings Have Been Compared Using	Control	Statements	Maintainability, Reliability	Strings not intern( )-ed have been compared using the == operator.	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
incorrect Operator								
Static Method Assumed Dynamic	Control	Statements	Maintainability, Reliability	Static method has been assumed to be selected dynamically.	●	○	⊙	●
Incorrect Overriding of Methods in Constructor	Structure, Control	Methods, Statements	Maintainability, Reliability	An overridden constructor contains a method that uses the subclasses fields but has not been initialized in that constructor.	●	○	●	●
Default Logical Value Assumed	Control	Methods, Statements	Maintainability, Reliability	A statement has assumed return of logical value without comparison.	●	○	●	●
A Reference to a Final Feature Misused	Control	Attributes, Statements	Maintainability, Reliability	A feature declared as final allows change of data values in an object because it is called by reference and not by value.	●	○	●	●
Expanded Type Overflow	Control	Statements	Reliability	An overflow without warning occurred with respect to a type such as int, long, float, or double.	●	○	●	●
Return Type Declared for Constructor	Structure	Method	Reliability	A constructor has been declared with a return type.	●	○	○	●
void Type Declared for Constructor	Structure	Method	Maintainability, Reliability	A constructor has been declared with a void return type.	●	○	○	●
The + Operator	Control	Statements	Maintainability, Reliability	The + operator has mistakenly been used by the system to imply concatenation when addition was intended or vice versa.	●	○	●	●
Array Problems	Control	Statements	Maintainability, Reliability	No space has been allocated for array. No objects assigned to each array location. The type of array object and type of array element incompatible.	●	○	⊙	●
Casting Over Non-expanded Types	Control	Statements	Maintainability, Reliability	Casting has been used to work over nonexpanded types (objects other than primitives).	●	○	⊙	●

Lange and Chaudron investigated to what extent designers detect consistency defects and to what extent defects cause different interpretations by different readers. The defects they investigated are listed in Table 25.

Table 25. *Defects by (Christian F. J. Lange & Chaudron, 2006; Christian F. J. Lange et al., 2006)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Message without Name (EnN)	Semantic	Sequences	Compilability	In sequence diagrams arrows representing messages exchanged by objects should be annotated with a name that describes the message.	●	○	⊙	●
Message without Method (EcM)	Structure	Sequences, Calls	Compilability	No correspondence between the message name and a provided method name.	●	○	⊙	●
Message in the wrong direction (ED)	Structure	Sequences, Calls	Compilability	This inconsistency occurs if there is a message from an object of class A	●	○	⊙	●

				to an object of class B but the method corresponding to the message is a member of class A instead of class B.				
Class not instantiated in SD (CnSD)	Structure	Sequences, Classes	Compilability	No class instantiation in a sequence diagram of a class that is defined in a class diagram of the model.	●	○	◎	●
Object has no Class in CD (CnCD)	Structure	Sequences, Classes	Compilability	This inconsistency occurs if there is an object in a sequence diagram and no corresponding class is defined in any class diagram.	●	○	◎	●
Use Case without SD (UCnSD)	Structure	Use cases, Sequences	Compilability, Conformance	A use case that is not illustrated by any sequence diagram.	●	○	◎	●
Multiple definitions of classes with equal names (Cm)	Semantic	Use cases, Sequences	Compilability, Maintainability	More than one class has the same name in a single model. The different classes may be defined in the same diagram or in different diagrams.	●	○	○	●
Method not called in SD (MnSD)	Structure	Use cases, Sequences	Maintainability	A method of a class is not called as a message in any sequence diagram.	●	○	◎	●

4.10 Error Patterns

The concept "error patterns" was used by Andy Longshaw and Eoin Woods to describe more managerial problems (Longshaw & Woods, 2004). These error patterns are intended to help with system-wide decisions about how to handle domain or technical errors. In the literature they are defined as follows:

- *"[error patterns] relate to the use of error generating, handling and logging mechanisms – particularly in distributed systems." (Longshaw & Woods, 2004)*
- *"[error patterns] provide a landscape in which sensible and consistent decisions can be made about when to raise errors, what types of error to raise, how to approach error handling and when and where to log errors." (Longshaw & Woods, 2004)*

In the following table we list error patterns that were found in the literature. The main large collection of error patterns was collected by (Longshaw & Woods, 2004, 2005).

Table 26. *Error Patterns by (Longshaw & Woods, 2004, 2005)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Big Outer Try Block	Control	Classes, Statements	Maintainability	Exceptional conditions are rarely anticipated in the design of the system and should be handled before the crash of the system.	●	●	○	●
Log at Distribution Boundary	Semantic	Classes	Maintainability	Propagating technical errors between system tiers results in error details ending up in locations (such as end-user PCs) where they are difficult to access and in a context far removed from that of the original error.	●	●	○	◎
Log Unexpected Errors	Semantic	Classes	Maintainability	Standard or common exceptions should be handled separately from unexpected or rare ones.	●	◎	○	◎

Make Exceptions Exceptional	Semantic	Classes	Maintainability	Exceptions are used to indicate expected error conditions occurring - calling code becomes much more difficult to understand.	●	⭘	○	⭘
Split Domain and Technical Errors	Semantic		Maintainability	Missing differentiation between "domain errors" and "technical errors". Technical errors (e.g., DB problems) must be handled while domain errors (e.g., Missing customer name) can be ignored.	●	●	○	⭘
Unique Error Identifier	Control	Classes, Statements	Maintainability	If an error on one tier in a distributed system causes knock-on errors on other tiers you get a distorted view of the number of errors in the system and their origin.	●	●	○	⭘
Hide Technical Detail from Users	Semantic	Classes	Maintainability, Usability	The technical details of errors may cause unnecessary concern and support overhead.	●	●	⭘	⭘
Ignore Irrelevant Errors	Semantic	Classes	Maintainability	Technical errors or exceptions do not denote a real problem and so reporting them can just be confusing or irritating for support staff.	●	⭘	○	●
Single Type for Technical Errors	Structure	Classes, Inheritance	Maintainability, Reliability	Exception Hierarchy with far too few Classes.	●	⭘	○	●

4.11 Fault Patterns

The concept "fault patterns" is similar in name to design patterns but is more similar to coding or design defects. Fidel Nkwocha and Sebastian Elbaum used the term to describe problems in end-user programming environments such as Matlab while Alexander used it for inheritance and polymorphism problems. These fault patterns represent problematic parts of the software system that produce faulty or uncompilable code. In the literature they are defined as follows:

- *"Fault patterns are code idioms that may constitute faults." (Nkwocha & Elbaum, 2005)*
- *"[Fault patterns] are useful because they indicate the possible presence of faults that result from the use of inheritance and polymorphism." (Alexander et al., 2002)*

Table 27. *Fault Patterns (in Matlab) by (Nkwocha & Elbaum, 2005)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Definition without a Usage: Def!Used	Control	Attributes, Statements	Maintainability, Efficiency	A variable is declared and allocated, but never used – i.e., memory was unnecessarily allocated.	●	⭘	○	●
Usage without a previous Definition: Used!Def	Control	Attributes, Statements	Maintainability, Reliability, (Usability)	A variable is being utilized before its definition – resulting in late discovery in an interpreter.	●	⭘	⭘	●
File may not get Closed: FOpened!Close	Control	Statements	Maintainability, Reliability	A "file opening" statement is not followed by a corresponding "file closing" statement – sometimes because open streams may lead to undefined behavior, may claim resources for longer than necessary, or may just cause failures if other operations are performed (e.g., open,	●	⭘	⭘	●

				share).				
Unmatched Returned Values: UReturns	Control	Statements	Maintainability, Reliability	A function returns an unexpected number of output or returned values were ignored.	●	○	○	●
Unreachable Functions: Function!Used	Structure	Statements	Maintainability	Functions that are not invoked throughout the program.	●	○	○	●
Switch without a Default: Switch!Otherwise	Control	Statements	Maintainability, Reliability	A switch statement without a default Clause – i.e., the default behavior is missing when the switch values do not occur.	●	○	●	●
Improper Exception Handling: Try!Catch	Control	Statements	Maintainability, Reliability	A try without its corresponding catch	⊙	○	●	●
Likely Infinite Loop: InfLoop	Control	Statements	Maintainability, Reliability	A looping structure with no obvious exit strategy.	●	○	●	●

Table 28. *Fault Patterns by (Alexander et al., 2002)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
ITU Inconsistent Type Use (context swapping)	Structure	Classes, Inheritance, Methods	Maintainability, Reliability	A descendant class does not override any inherited method - thus, there can be no polymorphic behavior.	●	○	⊙	●
SDA State Definition Anomaly (possible post-condition violation)	Structure, Semantic	Classes, Inheritance, Methods	Maintainability, Reliability	Refining methods implemented in the descendant must leave the ancestor in a state that is equivalent to the state that the ancestor's overridden methods would have left the ancestor in.	⊙	○	●	●
SDIH State Definition Inconsistency (due to state variable hiding)	Structure	Classes, Inheritance, Attributes	Maintainability, Reliability	A local variable is introduced to a class definition where the name of the variable is the same as an inherited variable.	⊙	○	●	●
SDI State Defined Incorrectly (possible post-condition violation)	Structure, Semantic	Classes, Inheritance, Attributes	Maintainability, Reliability	If the computation performed by an overriding method isn't semantically equivalent to the overridden method, then subsequent state dependent behavior in the ancestor will likely be affected - the externally observed behavior of the descendant will be different from the ancestor.	●	○	●	●
IISD Indirect Inconsistent State Definition	Structure, Semantic	Classes, Inheritance, Attributes	Maintainability, Reliability	A descendant adds an extension method that defines an inherited state variable – resulting in a data flow anomaly by having an effect on the state of the ancestor that is not semantically equivalent to the overridden method.	●	○	●	●

4.12 Flaws

One of the commonly used umbrella terms for smells and antipatterns on the software design level is "design flaw". While only few problems are themselves called or categorized as flaws other problems are typically subsumed with this term. In general, flaws are problems that are associated with one or more design principle or heuristic. In the literature they are defined as follows:

- *"The structural characteristic of a design entity or design fragment that expresses a deviation fen set of criteria typifying the high-quality of a design" (Marinescu, 2002)*

The flaw concept is used to describe problems that reduce the quality of a software system (mostly on the structural design level).

Beside the flaws on the code or design levels many other problems were described using the flaw metaphor. Today, we have flaws on different abstraction layers, for development phases, or technologies such as design flaws (Marinescu, 2002) or security flaws (Petroni & Arbaugh, 2003).

Table 29. *Design Flaws by (Marinescu, 2002)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Feature Envy	Structure	Classes, Calls	Maintainability	A method that is more interested in data of another class than the one of its own.	●	○	⊙	⊙
God Method	Structural	Classes, Methods	Maintainability, Portability	A method that centralizes the functionality in a class.	●	○	●	●
Data Class	Structure	Classes, Attributes, Methods	Maintainability	Classes that do almost exclusively store information for other classes. Optionally, these classes have getter and setter methods for the attributes.	●	●	○	●
God Class	Structural	Classes, Associations	Maintainability, Portability	Classes with too many functionality and associations to other classes.	●	○	⊙	●
Shotgun Surgery	Historic	Versions, Classes	Maintainability, Portability	Several classes are changed in a group every time a specific kind of change is to be made.	●	○	○	○
God Package	Structural	Packages, Associations	Maintainability, Portability	Packages with too many client packages.	●	○	⊙	●
Wide Subsystem Interface	Structure	Subsystem	Maintainability	The interface to a subsystem is too large (too many open packages and classes)	●	○	○	●
Lack of Bridge	Structure	Classes	Maintainability	Absence of the Bridge Pattern	●	○	○	●
Lack of Strategy	Structure	Classes	Maintainability	Absence of the Strategy Pattern	●	○	○	●

Table 30. *Design Flaws by (Marinescu & Lanza, 2006)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
God Class	Structural	Classes, Associations	Maintainability, Portability	Classes with too many functionality and associations to other classes.	●	○	⊙	●
Feature Envy	Structure	Classes, Calls	Maintainability	A method that is more interested in data of another class than the one of its own.	●	○	⊙	⊙
Data Class	Structure	Classes, Attributes, Methods	Maintainability	Classes that do almost exclusively store information for other classes. Optionally, these classes have getter and setter methods for the attributes.	●	●	○	●
Brain Method	Structural	Classes, Methods	Maintainability, Portability	A method that centralizes the functionality in a class.	●	○	●	●

Brain Class	Structural	Classes, Methods	Maintainability, Portability	A class that tends to accumulate an excessive amount of intelligence, usually in the form of several methods affected by Brain Method.	●	○	◎	●
Significant Duplication	Semantic	Methods, Statements	Maintainability, Reliability	Identical code passages are distributed over the whole system	●	○	●	○
Intensive Coupling	Structural	Methods, Calls	Maintainability, Portability	A method is tied to many other close operations in the system	●	○	◎	●
Dispersed Coupling	Structural	Methods, Calls	Maintainability, Portability	A method is tied to many other distributed operations in the system	●	○	◎	●
Shotgun Surgery	Historic	Versions, Classes	Maintainability, Portability	Several classes are changed in a group every time a specific kind of change is to be made.	●	○	○	○
Refused Parent Bequest	Structure	Classes, Attributes, Methods	Maintainability	Subclasses that inherit attributes and methods that they do not use.	●	○	◎	◎
Tradition Breaker	Structure	Classes Inheritance	Maintainability	The interface of a class breaks the inherited "tradition", e.g., has an excessive increase.	●	○	◎	●

4.13 Heuristics

Beside the explicit description of problems the literature has a large corpus of heuristics and characteristics that should be applied – esp. on the object-oriented design level. While many of them are descriptions of positive practices or (code) structures / designs (e.g., "Minimize Fanout in a class") several of them are negations of negative practices (e.g., "Do not create god classes"). Similar to patterns we can not unconditionally use or invert a heuristic and find a bad or worst practice. However, in the case of these negated heuristics it is possible. In general, bad heuristics are problems that are associated with one or more design principle. In the literature they are defined as follows:

- *"[Heuristics] are meant to serve as warning mechanisms which allow the flexibility of ignoring heuristic as necessary" (Riel, 1996b)*
- *"[Heuristic is] A small and legible piece of design expertise that delivers experience from the expert to the novice in the most effective manner." (Gibbon, 1997)*
- *"A heuristic is a rule of thumb. It is an advice on how to use design techniques in order to solve design problems. It provides guidelines for finding appropriate solutions." (Grotehen, 2001)*

The bad heuristic concept is used to describe problems that reduce the quality of a software system (mostly on the design level).

Beside the bad heuristics on the code or design levels many other problems were described using the heuristic metaphor. Today, we have bad heuristics on different abstraction layers, for development phases, or technologies such as heuristics for object-oriented systems (Riel, 1996b) (Shadrin, 2005) and interactive systems (Cockton & Gram, 1996).

In the following sections we will list most of these bad heuristics that were found in the literature survey. The first larger collection of bad heuristics was collected by Arthur J. Riel:

Table 31. *Heuristics by (Riel, 1996b)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
#2.1: All data should be hidden within its class.	Structure	Classes	Maintainability	The classes gives access on far too many information	●	◎	○	●
#2.2: Users of a class must be dependent on its public interface, but a class should not be dependent on its users.	Structure	Classes, Association	Maintainability, Portability	A class depends on its users.	●	○	◎	◎
#2.3: Minimize the number of messages in the protocol of a class.	Structure	Classes, Methods	Maintainability, Portability	Too many methods	●	○	○	●
#2.4: Implement a minimal public interface which all classes understand	Semantic	Classes, Inheritance, Methods	Maintainability, Portability	Too many similar methods in unrelated classes (e.g. operations such as copy (deep versus shallow), equality testing, pretty printing, parsing from a ASCII description, etc.).	●	○	○	○
#2.5: Do not put implementation details such as common-code private functions into the public interface of a class.	Structural	Classes, Methods	Maintainability, Portability	The complexity of the class interface is too big. Methods in the class interface that are not used. Methods that are used by other methods (i.e., common code) in the interface.	●	○	◎	●
#2.6: Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.	Structure	Classes, Methods	Maintainability, Portability	Methods in a class interface that cannot be used.	●	○	◎	●
#2.7: Classes should only exhibit nil or export coupling with other classes	Structure	Classes, Methods	Maintainability, Portability	A class should only use operations in the public interface of another class or has nothing to do with that class.	●	○	◎	●
#2.8: A class should capture one and only one key abstraction.	Structure, Semantic	Classes, Methods, Interfaces	Maintainability	A class with a large number of public responsibilities (e.g., interfaces). Multi-Noun class names.	●	◎	○	●
#2.9: Keep related data and behavior in one place.	Structure	Classes, Methods, Statements	Maintainability	Classes that dig data out of other classes using getter-methods	●	◎	◎	●
#2.10: Spin off non-related information into another class (i.e. noncommunicating behavior).	Structure	Classes, Methods, Statements	Maintainability	A subset of methods works on a subset of attributes (i.e. different responsibilities)	●	○	●	●
#2.11: Be sure the abstraction that you model are classes and not simply the roles objects play.	Structure, Semantic	Classes, Methods, Names	Maintainability	A class that incorporate two or more behaviors based on a role (e.g., Person'). Classes with different names but similar or identical behavior (e.g., Mother and Father).	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
#3.1: Distribute system intelligence horizontally as uniformly as possible	Structure	Classes, Methods, Statements	Maintainability	The top level classes in a design should share the work uniformly.	●	○	●	●
#3.2: Do not create god classes/objects in your system.	Structural, Semantic	Classes, Associations, Names	Maintainability, Portability	Classes with too many functionality and associations to other classes. Be very suspicious of an abstraction whose name contains Driver, Manager, System, or Subsystem.	●	○	◎	●
#3.3: Beware of classes that have many accessor methods defined in their public interface,	Structural	Classes, Methods, Names	Maintainability, Portability	Classes with too many functionality and associations to other classes. Many accessor methods imply that related data and behavior are not being kept in one place.	●	○	◎	●
#3.4: Beware of classes which have too much non-communicating behavior	Structure	Classes, Methods, Statements	Maintainability	Methods which operate on a proper subset of the data members of a class. God classes often exhibit lots of non-communicating behavior.	●	○	●	●
#3.5: The model should never be dependent on the interface. The interface should be dependent on the model.	Structure	Classes, Methods, Statements	Maintainability	In applications which consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.	●	○	●	●
#3.6: Model the real world whenever possible.	Semantic	Names	Maintainability	Names do not match the real world	◎	○	○	●
#3.7: Eliminate irrelevant classes from your design.	Structure	Classes, Attributes, Methods	Maintainability	Classes that do almost exclusively store information for other classes. Optionally, these classes have accessor or print methods.	●	●	○	●
#3.8: Eliminate classes that are outside the system.	Structure	Classes, Methods, Statements	Maintainability	A class with methods that aren't used or required.	●	○	◎	●
#3.9: Do not turn an operation into a class.	Semantic	Names	Maintainability	Be suspicious of any class whose name is a verb or derived from a verb. Especially those which have only one piece of meaningful behavior (i.e. do not count sets, gets, and prints).	●	○	○	●
#3.10: Agent classes are often placed in the analysis model of an application.	Structure	Classes, Methods, Statements	Maintainability	A decoupling class with more methods than data that uses a class with more data than method and is used by a third class.	●	○	◎	●
#4.1: Minimize the number of classes with which another class collaborates.	Structure, Control	Classes, Methods, Calls, Statements	Maintainability	Too many classes are linked to this class by an extensive network of data or control flows.	●	○	◎	●
#4.2: Minimize the number of message sends between a class and its colla-	Structure, Control	Classes, Methods, Calls, Statements	Maintainability	Too many messages are send between this class and a collaborator	●	○	●	◎

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
borator.								
#4.3: Minimize the amount of collaboration between a class and its collaborator, i.e. the number of different messages sent.	Structure, Control	Classes, Methods, Calls, Statements	Maintainability	Too many different messages are send between this class and a collaborator	●	○	●	◎
#4.4: Minimize fanout in a class	Structure, Control	Classes, Methods, Calls, Statements	Maintainability	The product of the number of messages defined by the class and the messages they send is too high.	●	○	◎	●
#4.5: If a class contains objects of another class then the containing class should be sending messages to the contained objects	Structure, Control	Classes, Attributes, Methods, Calls, Statements	Maintainability	The containment relationship should always imply a uses relationship.	●	○	◎	●
#4.6: Most of the methods defined on a class should be using most of the data members most of the time.	Structure, Control	Classes, Attributes, Methods, Calls, Statements	Maintainability	Attributes are not (enough) used by the methods in a class.	●	○	◎	●
#4.7: Classes should not contain more objects than a developer can fit in his or her short term memory. A favorite value for this number is six.	Structure	Classes, Attributes	Maintainability	Too many Objects in a class	●	○	○	●
#4.8: Distribute system intelligence vertically down narrow and deep containment hierarchies.	Structure, Control	Classes, Inheritance, Methods, Statements	Maintainability	The classes in an inheritance hierarchy should share the work uniformly.	●	○	◎	●
#4.9: When implementing semantic constraints, it is best to implement them in terms of the class definition.	Control	Classes, Attributes	Maintainability	Semantic constraints on a class (instantiation) not within the constructor.	◎	○	◎	●
#4.10: When implementing semantic constraints in the constructor of a class, place the constraint test in the constructor as far down a containment hierarchy as the domain allows.	Control	Classes, Inheritance, Methods, Statements	Maintainability	Unnecessary constraints in a class that belongs to a subclass	◎	○	◎	●
#4.11: The semantic information on	Structure	Classes, Attributes,	Maintainability	Semantic information encoded in the same class it is needed	◎	○	◎	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
which a constraint is based is best placed in a central third-party object when that information is volatile.		Methods, Statements						
#4.12: The semantic information on which a constraint is based is best decentralized among the classes involved in the constraint when that information is stable.	Control	Classes, Inheritance, Methods, Statements	Maintainability	Semantic information encoded in the classes that needs it.	◎	○	◎	◎
#4.13: A class must know what it contains, but it should never know who contains it.	Structure	Classes, Attributes, Methods	Maintainability	The contained class should not have a reference on the containing one.	●	○	○	●
#4.14: Objects which share lexical scope should not have uses relationships between them.	Structure	Classes, Attributes, Calls, Statements	Maintainability	Classes shared as objects in other classes (i.e., its fields) should not use each other.	●	○	○	◎
#5.1: Inheritance should only be used to model a specialization hierarchy.	Structure, Control	Classes, Inheritance, Calls, Statements	Maintainability	Inheritance is used instead of containment.	◎	○	◎	●
#5.2: Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.	Structure, Control	Classes, Inheritance, Calls, Statements	Maintainability	Base classes with access to or containing a derived class.	●	○	◎	●
#5.3: All data in a base class should be private, i.e. do not use protected data.	Structure, Control	Classes, Attributes	Maintainability	Non-private attibutes in a base class	●	○	○	●
#5.4: Theoretically, inheritance hierarchies should be deep, i.e. the deeper the better.	Structure	Classes, Inheritance	Maintainability	The inheritance tree is not depth enough.	●	○	○	◎
#5.5: Pragmatically, inheritance hierarchies should be no deeper than an average person can keep in their short term memory.	Structure	Classes, Inheritance	Maintainability	The inheritance tree is too depth. A popular value for this depth is six.	●	○	○	◎
#5.6: All abstract	Structure	Classes,	Maintainability	An abstract class must have children.	●	○	○	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
classes must be base classes.		Inheritance						
#5.7: All base classes should be abstract classes.	Structure	Classes, Inheritance	Maintainability	An abstract class inherits from another base class.	●	○	○	●
#5.8: Factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy.	Structure, Semantic	Classes, Inheritance, Attributes, Methods	Maintainability	Commonalities between all related classes should be shared in a common ancestor.	●	○	○	⊙
#5.9: If two or more classes only share common data (no common behavior) then that common data should be placed in a class which will be contained by each sharing class.	Structure, Semantic	Classes, Inheritance, Attributes, Methods	Maintainability	Data commonalities between some related classes should be shared in a contained class.	●	⊙	○	⊙
#5.10: If two or more classes have common data and behavior (i.e. methods) then those classes should each inherit from a common base class which captures those data and methods.	Structure, Semantic	Classes, Inheritance, Attributes, Methods	Maintainability	Commonalities between unrelated classes should be shared in a common ancestor.	●	○	○	⊙
#5.11: If two or more classes only share common interface (i.e. messages, not methods) then they should inherit from a common base class only if they will be used polymorphically.	Structure, Semantic	Classes, Inheritance, Usage	Maintainability	Classes implementing an interface that is not needed or used.	●	○	○	●
#5.12: Explicit case analysis on the type of an object is usually an error.	Structure, Control	Classes, Inheritance, Statements	Maintainability, Reliability	Case or Switch statements are used to differentiate between different classes.	⊙	○	●	⊙
#5.13: Explicit case analysis on the value of an attribute is often an error.	Structure, Control	Classes, Atrributes, Statements	Maintainability, Reliability	Case or Switch statements are used to differentiate between different attribute values (e.g., states).	⊙	○	●	⊙
#5.14: Do not model the dynamic semantics of a class through the use of the inheritance relationship.	Structure, Control	Classes, Statements	Maintainability, Reliability	An attempt to model dynamic semantics with a static semantic relationship will lead to a toggling of types at runtime.	⊙	○	●	⊙

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
#5.15: Do not turn objects of a class into derived classes of the class.	Structure	Classes	Maintainability	Be very suspicious of any derived class for which there is only one instance.	●	○	◎	●
#5.16: If you think you need to create new classes at runtime, take a step back and realize that what you are trying to create are objects. Now generalize these objects into a class.	Control	Classes	Maintainability	Creation of classes at runtime	◎	○	●	●
#5.17: It should be illegal for a derived class to override a base class method with a NOP method, i.e. a method which does nothing.	Structure	Class, Inheritance	Maintainability	A method is overridden with an empty method.	●	○	◎	●
#5.18: Do not confuse optional containment with the need for inheritance, modeling optional containment with inheritance will lead to a proliferation of classes.	Structure	Class, Inheritance	Maintainability	Containment modeled as inheritance	◎	○	○	●
#5.19: When building an inheritance hierarchy try to construct reusable frameworks rather than reusable components.	Semantic	Classes, Inheritance, Names	Portability	Inheritance hierarchy could be more general to fit the domain instead of the system.	◎	○	○	◎
#6.1: If you have an example of multiple inheritance in your design, assume you have made a mistake and prove otherwise.	Structural	Classes, Inheritance	Maintainability	A class inherits from two or more classes (i.e., multiple inheritance)	◎	○	○	●
#6.2: Whenever there is inheritance in an object-oriented design ask yourself two questions: 1) Am I a special type of the thing I'm inheriting from? and 2) Is the thing I'm inheriting from part of me?	Semantic	Classes, Inheritance, Names	Maintainability	Inheritance is not based on a specialization concept (e.g., is-a)	●	◎	○	●
#6.3: Whenever you have found a multiple inheritance	Structural	Classes, Inheritance	Maintainability	One base class is derived from another base class above a class	●	○	○	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
relationship in an object-oriented design be sure that no base class is actually a derived class of another base class, i.e. accidental multiple inheritance.								
#7.1: When given a choice in an object-oriented design between a containment relationship and an association relationship, choose the containment relationship.	Structural	Classes, Calls	Maintainability	Try to change associations into containments, if possible.	●	○	○	●
#8.1: Do not use global data or functions to perform bookkeeping information on the objects of a class, class variables or methods should be used instead.	Structure, Control	Attributes, Calls	Maintainability	An externally declared variable that is referenced within a class but is not an attribute within the class.	●	◎	◎	●
#9.1: Object-oriented designers should never allow physical design criteria to corrupt their logical designs.	Semantic	Classes	Maintainability	Design should be understandable and not necessarily 100% perfect regarding the real world	◎	○	○	◎
#9.2: Do not change the state of an object without going through its public interface.	Structural	Classes	Maintainability	The classes gives access on attributes responsible to hold the state	●	◎	○	●

Table 32. *Heuristics by (Gibbon, 1997)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
CC1 Limit the number of methods per class	Structure	Classes, Methods, Statements	Maintainability, Portability	A class with far too many methods, attributes, and consequently responsibilities.	●	○	○	●
CC2 Limit the number of attributes per class	Structure	Classes, Attributes	Maintainability	A class with far too many attributes.	●	○	○	●
CC3 Limit the messages that an object can receive	Structure, Control	Classes, Usage	Maintainability	A class with far too many incoming messages (users).	●	○	◎	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
CC4 Minimise complex methods	Control	Methods	Maintainability	A class with too much complexity (e.g., too many complex methods)	●	○	◎	●
CC5 Limit enabling mechanisms that breach encapsulation	Structural	Classes, methods	Maintainability	The classes gives access on far too many information via methods	●	◎	○	●
CC6 Hide all implementation details	Structural	Classes	Maintainability	The classes gives access on far too many information	●	◎	○	●
CU1 Limit the number of collaborating classes	Structural	Methods, Calls	Maintainability, Portability	Too many classes are coupled among each other.	●	○	◎	◎
CU2 Restrict the visibility of interface collaborators	Structural	Classes	Maintainability	The interface gives access on far too many information	●	◎	○	●
CA1 The aggregate should limit the number of aggregated	Structural	Classes, Attributes	Maintainability	Too many aggregated classes	●	◎	○	●
CA2 Restrict access to aggregated by clients	Structural, Control	Classes, Calls	Maintainability	Too many aggregated attributes are accessed by clients	●	◎	◎	●
RA1 Aggregation hierarchies should not be too deep	Structure	Classes, Attributes, Inheritance	Maintainability	The inheritance tree for classes that are aggregated is too depth.	●	○	○	◎
RA2 The leaf nodes in an aggregation hierarchy should be small, reusable and simple	Structure	Classes, Inheritance, Methods	Maintainability, Portability	A leaf class with far too many methods, attributes, and consequently responsibilities.	●	○	○	●
RA3 Stability should descend the hierarchy from rich aggregates to their building blocks	Structure, Historic	Classes, Inheritance, Attributes, Versions	Maintainability, Portability	The stability of an aggregation hierarchy relies upon the stability of its leaf nodes and the extent to which its uppermost aggregates have encapsulated them.	●	○	○	○
CI1 Limit the use of multiple inheritance	Structure	Classes, Inheritance	Maintainability	The amount of classes with multiple inheritance should be 0.	●	○	○	●
CI2 Prevent overgeneralisation of the parent class	Structure	Classes, Methods, Statements	Maintainability	A base class that isn't doing much.	●	○	◎	●
RI1 The inheritance hierarchy should not be too deep	Structure	Classes, Inheritance	Maintainability	The inheritance tree is too depth.	●	○	○	◎
RI2 The root of all inheritance hierarchies should be abstract	Structure	Classes, Inheritance	Maintainability	A base class should be abstract.	●	○	○	●
RI5 Strive to make as many intermediary nodes as possible abstract.	Structure	Classes, Inheritance	Maintainability	An abstract class inherits from another base class.	●	○	○	●
RI6 Stability should ascend the inheritance hierarchy	Structure, Historic	Classes, Inheritance, Versions	Maintainability, Portability	The stability of a hierarchy relies upon the stability of its base classes.	●	○	○	○
RI7 Inheritance is a specialisation hie-	Semantic	Classes, Inheritance,	Maintainability	Inheritance is not based on a specialization concept (e.g., is-a)	●	◎	○	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
rarchy		Names						

Table 33. *Heuristics by (Grotehen, 2001)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
A class in a containment hierarchy should only depend from its child classes	Structure, Control	Classes, Inheritance, Calls, Statements	Maintainability	A class should neither depend from its container or from one of its siblings. A class should not depend from classes below its siblings.	●	○	◎	●
Every attribute should be hidden within its class	Structure	Classes, Attributes, Calls	Maintainability	Do not use attributes in the public interface of a class.	●	○	◎	◎
A client-server dependency between two classes should not Lead to dependencies from the server to the client	Structure	Classes, Attributes, Methods, Calls	Maintainability	Do not use attributes or methods in the public interface of a client class in its server class. Do not inherit from a client class. Do not send messages to instances of a derived class.	●	○	◎	◎
Avoid dependencies from database classes to their clients	Structure	Layers, Calls	Maintainability, Portability	Avoid relationships from database classes to classes outside the database. Use callback functions or event mechanisms if communication from a database to its clients is required.	●	○	◎	●
A class should capture one and only one key abstraction with All its information and all its behaviour	Structure, Semantic	Classes, Methods, Interfaces	Maintainability	Do not distribute knowledge about a key abstraction among many classes. Do not model different key abstractions in a single class.	●	◎	○	●
Do not create unnecessary classes to model roles	Structure, Semantic	Classes, Methods, Names	Maintainability	Classes that are too similar (and probably related). Model only one class for an entity with different roles and provide the role information in other ways e.g. in state attributes.	●	○	●	●
Avoid pure accessor methods	Structural	Classes, Methods, Names	Maintainability, Portability	Try to minimize the number of methods which only return or change an attribute of their class. Instead of pure accessor methods use methods which implement some interesting behavior of the instance.	●	○	◎	●
Avoid additional relationships from base classes to their Derived classes	Structure, Control	Classes, Inheritance, Calls, Statements	Maintainability	Avoid associations and using relationships that lead to a dependency from a base class to its derived class.	●	○	◎	●
Avoid classes with properties implying redundancies	Semantic	Classes, Methods, Attributes	Maintainability, Reliability	Data is stored redundantly in multiple classes. This heuristic is similar to 3nf (Third normal form in Databases).	●	●	○	●
Avoid multivalued dependencies	Semantic	Attributes	Maintainability	Eliminate every multivalued dependency.	◎	◎	○	●
Convert associa-	Structure	Associations	Maintainability	Replace loose relationships by rela-	●	○	◎	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
tions, and uses relationships in the Strongest containment relationship wherever possible				tionship that restrict visibility. Avoid many loose relationships. The application of this heuristic converts a given class hierarchy into a narrow and deep containment hierarchy.				
Avoid contained instances that have to be modified Concurrently	Structure, Control	Classes, Methods, Statements	Maintainability, Efficiency	Avoid a class specification where different contained instances have to be modified by concurrent transactions (e.g., because of (static) class variables).	●	○	○	●
All properties of the base class interface must be usable in Instances of its derived classes in every location where a base Class instance is expected	Structure	Class, Inheritance	Maintainability	The interface of a derived class should fully implement its base class interface. If a base class instance is expected, no additional properties of a derived class should be needed.	●	○	◎	●
Common properties of instances should be defined in a single Location	Semantic, Structure	Classes, Inheritance, Attributes, Methods	Maintainability	Avoid properties that have the same meaning and are defined in different locations. Move common properties in derived classes to the base class.	●	○	○	◎
Instable classes should not be base classes	Structure, Historic	Classes, Inheritance, Attributes, Versions	Maintainability, Portability	Classes that depend on many other classes should not be base classes. Classes that are instanciable should not be base classes	●	○	○	●
Do not misuse inheritance for sharing attributes	Structure, Semantic	Classes, Inheritance, Attributes, Methods	Maintainability	Avoid using inheritance if you intend to share only the attributes of the inherited base class among the derived classes. Use association of aggregation of a shared instance for sharing attributes.	●	◎	○	◎
The overloading should define only differences to the overloaded method	Structure	Classes, Methods, Statements	Maintainability	Analyze exactly the differences in both methods than describe these differences in the overloading method.	●	○	◎	●
Avoid case analysis on properties of instances	Structure, Control	Classes, Atrributes, Statements	Maintainability, Reliability	Avoid case analysis on attributes of an instance. Avoid case analysis on attributes of an instance, which influence its behavior.	◎	○	●	◎
Prefer typing by attribute before typing by inheritance	Structure, Control	Classes, Inheritance, Statements	Maintainability, Reliability	If the differences are small the differing instances should be modeled in the same class. Attributes should be used to model the differences.	◎	○	●	◎
A method should use only classes of attributes of its class, classes of its parameters, or classes of instances locally created.	Strucutre, Control	Classes, Methods, Calls	Maintainability	Use of too many foreign classes. This is an application of the law of Demeter. The implementers of a method should only use this restricted set of classes in the method implementation.	●	○	◎	●

4.14 Illnesses

The medicine-based term illness was used by Hawkins in the book “” to describe problems on the management level (Hawkins, 2003). These software illnesses represent problematic parts

of the software system that seem wrong, complicated, or cumbersome to an experienced developer. In general, illnesses are abstract problems that mostly cannot be pinpointed directly in the source code or a software model. In the literature they are defined as follows:

- *"[Illnesses are] a metaphor for describing programming errors"* (Hawkins, 2003)

While some illnesses are general problems, e.g., in the development process they do consist of several smaller individual problems that are directly associated with the architecture, design, or code. Illnesses such as "NIH syndrome" were not excluded even if the refusal of external code is more a productivity problem (and has no negative of the quality of the resulting system per se).

Table 34. *Illnesses by (Hawkins, 2003)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Premature Optimization	Semantic	Statements, Comments	Maintainability, Portability	Optimizing too early in the development process, e.g., by uncommunicative code.	●	○	⭘	⭘
CAP Epidemic	Semantic	Methods, Statements, Comments	Maintainability, Portability	Duplicating code or comments using Copy And Paste (CAP)results in distributed changes and bugs.	●	○	●	○
NIH Syndrome	Semantic	External components	*(Productivity)*	Fear of (re-)using external code or libraries.	⭘	○	○	⭘
Complexification	Semantic	Classes, Methods	Efficiency	Making a solution more complex than it has to be	⭘	○	⭘	⭘
Over Simplification	Structure	Classes, Methods, Statements	Maintainability	Making a solution too simple than it has to be and creating too many small methods and classes.	●	○	⭘	●
Docuphobia	Semantic	Notes / Comments	Maintainability	Writing too few or uncommunicative comments and documentation	●	○	⭘	●
"i"	Semantic	Comments, Names	Maintainability	Usage of uncommunicative names for variables or redundant comments	●	○	⭘	●
Hardcode	Semantic	Attributes, Names	Maintainability, Portability	Hard-coded numbers and strings in the code	●	○	○	●
Brittle Bones	Semantic	Methods, Statements	Maintainability, Reliability	Applications build on buggy libraries, instable cores, or brittle frameworks with missing, unused, overly different, or overly complex features.	⭘	○	⭘	⭘
Requirement Deficiency	Semantic, Structure	Requirement document	Maintainability, Reliability	Unfinished, incomplete, vague, abstract, or large requirements	○	○	○	⭘
Myopia	Semantic	Classes, Methods	Maintainability, Reliability, Portability	Unfinished problems with sub-optimal solutions (quick-fixes, workarounds, etc.).	⭘	○	⭘	⭘

4.15 Metric Thresholds

A relatively basic concept of defects is captured with the "threshold" concept. While metrics are often used to assess the software quality or to predict the project development sometimes thresholds are used to describe concrete problems. Lorenz and Kidd used this term in their book "Object-oriented software metrics" to describe problems in object-oriented software system that should be removed (Lorentz & Kidd, 1994). In the literature they are defined as follows:

- *"A measurement value that has been determined through project experiences to be significant in terms of desirable or undesirable designs, with some margin of error." (Lorentz & Kidd, 1994)*

Table 35. *Metric Thresholds by (Lorentz & Kidd, 1994)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
(Optimum) Number of Key Classes	Structure	Classes	Maintainability	A system should consist of 20-40% key classes (i.e., classes central to the business domain)	●	○	○	⊙
(Optimum) Number of support classes	Structure	Classes	Maintainability	A system should consist of one to three times as much support classes (i.e., classes providing basic service or interface capabilities)	●	○	○	⊙
(Optimum) Number of subsystems	Structure	Classes	Maintainability	There shouldn't be too many subsystems (i.e., collections of classes that supports a set of end-user functions)	●	○	○	⊙
(Optimum) Number of message sends	Structure	Method, Calls	Maintainability, Portability	There shouldn't be more than nine messages send by a method.	●	○	●	●
(Optimum) Number of statements	Structure	Method, Statements	Maintainability	There shouldn't be more than seven statements in a method.	●	○	●	●
(Optimum) Lines of Code	Structure	Method, Statements	Maintainability	There shouldn't be more than six (Smalltalk) or 24 (C++) lines of code in a method.	●	○	●	●
(Optimum) Method complexity	Control	Method, Statements	Maintainability	Methods shouldn't have a complexity over 65 (based on defined weights).	●	○	●	●
(Optimum) Number of public instance methods in a class	Structure	Class, Methods	Maintainability	There shouldn't be more than 20 public instance methods in a class.	●	○	○	●
(Optimum) Number of instance methods in a class	Structure	Class, Methods	Maintainability	There shouldn't be more than 20 instance methods in a class (40 in UI classes).	●	○	○	●
(Optimum) Number of instance variables in a class	Structure	Class, Attributes	Maintainability, Portability	There shouldn't be more than 3 instance variables in a class (9 in UI classes).	●	○	○	●
(Optimum) Number of class methods in a class	Structure	Class, Methods	Maintainability	There shouldn't be more than 4 class methods (i.e., static) in a class.	●	○	○	●
(Optimum) Number of class variables in a class	Structure	Class, Attributes	Maintainability	There shouldn't be more than 3 class variables (i.e., static) in a class.	●	○	○	●
(Optimum) Class hierarchy nesting level	Structure	Class, Inheritance	Maintainability	The inheritance hierarchy level should be lower than 6.	●	○	○	●
(Optimum) Number of abstract classes	Structure	Class	Maintainability	A system should consist of 10-15% abstract classes	●	○	○	⊙
(Optimum) Use of inheritance	Structure	Class, Inheritance	Maintainability	The amount of classes with multiple inheritance should be 0.	●	○	○	●
(Optimum) Number of methods overridden by a subclass	Structure	Class, Inheritance	Maintainability	The amount of methods overridden should be less than 3.	●	○	○	●
(Optimum) Number of methods inherited by a subclass	Structure	Class, Inheritance	Maintainability	The amount of methods inherited should be high.	●	○	○	●
(Optimum) Number	Structure	Class, Inhe-	Maintainability	The amount of methods added	●	○	○	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
of methods added by a subclass		ritance		should be between 1 and 20 (depends on the inheritance level).				
(Optimum) Class cohesion	Structure	Class, Methods, Attributes, Calls	Maintainability	The message connections within a class and the use of instance variables.	●	○	○	●
System Global	Structure	Class, Inheritance	Maintainability	There should be at most only one system global, class variable, or pool dictionary	●	○	○	⊙
Average number of parameters per method	Structure	Methods, Parameters	Maintainability	The average amount of parameter per method should be less than 0.7	●	○	⊙	●
Use of friend functions	Structure	Class, Methods, Calls	Maintainability	The amount of friend classes should be 0.	●	○	⊙	●
Average number of comment lines per method	Structure	Methods, Statements, Comments	Maintainability	The average amount of comment lines per method should be less than 1.	●	○	⊙	●
Average number of commented methods	Structure	Methods, Statements, Comments	Maintainability	The average amount of commented methods should be between 65% and 100%.	●	○	⊙	●
Class Coupling	Structure	Class, Methods, Attributes, Calls	Maintainability	The message connections between classes via methods and instance variables.	●	○	○	●

4.16 Negative Patterns

Furthermore, the concept "Negative Pattern" is very similar to the antipattern concept – as antipatterns these patterns represent problematic parts of the software system that seem wrong, complicated, or cumbersome to an experienced developer. In general, these negative patterns are problems that are associated with one or more specific refactorings (i.e., concrete treatments) that might be applied to remove the pattern. In the literature they are defined as follows:

- *"[negative patterns] can be expressed in negative form: 'avoid XYZ' …[and] have a corresponding positive pattern: 'avoid XYZ, do PQR instead'" (Veryard, 2001)*

The concept of negative patterns is used to describe the experience and knowledge that was acquired by experts and have been proven beneficial. In the following sections we will list most of these negative patterns that were found in the literature survey. Several unexplained or non-product focused problems such as "Avoid single point of failure" were excluded from the list.

Table 36. *Negative Patterns collected by (Veryard, 2001)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Avoid GOTO (GOTO considered harmful)	Control	Statements	Maintainability, Reliability	There exists a goto statement within the body of a function.	●	○	●	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Avoid executing data	Control	Statements	Maintainability, Reliability, Functionality	Execution of (or jumps to) system parts that might have been data.	●	○	●	●
Avoid hard-coding data into program	Control	Statements	Maintainability, Reliability	Constant numbers or Strings that appear multiple times in the system	●	●	●	○
Cycles lead to deadlocks	Structure	Classes, Calls	Maintainability, Portability	There is a cycle in the call structure	●	●	●	○
Minimize Use of Interrupts	Control	Statements	Maintainability, Portability	Too many interrupts	⊙	○	⊙	●
Globals Considered Harmful	Structure, Control	Calls	Maintainability	There exists an externally declared variable that is referenced within a function but has not been passed in as a parameter.	●	○	⊙	●
Hyperspaghetti Objects and Subsystems	Structural	Classes, Methods, Calls	Maintainability, Reliability	Classes call many other classes and the coupling between classes or subsystems is high	●	○	⊙	●
Don't Interrupt an Interrupt	Control	Statements	Reliability	Interrupting an interrupt	⊙	○	⊙	●
Avoid inhibiting garbage collection	Control	Statements	Reliability	Changing the behavior of garbage collection	⊙	○	⊙	●
Avoid excessive initialization overhead	Control	Statements	Maintainability, Efficiency	Doing too much work upfront.	●	○	●	●
Explicit Invocation - Tight Coupling	Control	Statements	Maintainability, Portability	Components and Classes are coupled too tighly	●	○	○	●
Implicit Collaboration Protocols - Code Pollution	Control	Statements	Maintainability, Portability	Implicit (rather than explicit) reflection of time-ordered collaboration protocols.	●	○	○	●

4.17 Pitfalls

A relatively old concept “pitfall” was used often in the 80th and 90th of the last century to describe problems in different situations. Bruce F. Webster used this term in his book to describe problems in object-oriented software development and systems (Webster, 1995). These pitfalls represent problematic parts of the software system that seem wrong, complicated, or cumbersome to an experienced developer. In general, pitfalls are problems that are associated with one or more specific refactorings (i.e., concrete treatments) that might be applied to remove the pitfalls. In the literature they are defined as follows:

- *“[pitfalls] threaten to undermine the acceptance and use of object-oriented development before its promise can be achieved” (Webster, 1995)*
- *“A pitfall is code that compiles fine but when executed produces unintended and sometimes disastrous results” (Daconta et al., 2000; Daconta et al., 2003)*
- *“[pitfalls are] the knowledge of which will be useful in instructing new programmers and in developing tools to aid in multi-threaded programming.” (Choi & Lewis, 2000)*

The concept of pitfalls is used to describe the experience and knowledge that was acquired by experts and have been proven beneficial.

Beside the pitfalls on the object-oriented code or design levels many other problems were described using the pitfall metaphor. Today, we have pitfall on different abstraction layers, for

development phases, or technologies such as pitfalls in real-time systems (Stewart, 1999), java programs (Daconta et al., 2000), or multi-threaded systems (Choi & Lewis, 2000).

In the following sections we will list most of these pitfalls that were found in the literature survey. The first larger collection of pitfalls was collected by Bruce F. Webster:

Table 37. *Pitfalls by (Webster, 1995)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Confusing is-a, has-a and is-implemented-using relationships	Semantic	Inheritance, Names	Maintainability	Inheritance is not based on one specialization concept (e.g., is-a) or mixes several forms of inheritance in one inheritance tree.	●	⭘	○	⭘
Confusing interface inheritance with implementation inheritance	Semantic	Classes, Inheritance, Names	Maintainability, Reliability	Inheritance from interfaces and “real” classes is confused or changed by using own implemented methods.	●	○	⭘	●
Using Inheritance badly (Violate encapsulation)	Semantic	Classes, Inheritance	Maintainability	A base class is subclasses just to get access to its attributes and methods	●	○	○	●
Using Inheritance badly (Invert is-a by multiple inheritance)	Structural	Classes, Inheritance	Maintainability	A class inherits from two or more classes and generate a kind of superclass (by derivation)	⭘	○	○	●
Using Inheritance badly (Using multiple inheritance)	Structural	Classes, Inheritance	Maintainability	A class inherits from two or more classes (i.e., multiple inheritance)	⭘	○	○	●
Having base classes do too much or to little	Structural	Classes, Methods, Attributes	Maintainability	A (concrete) base class implements too much or too little behavior.	●	○	○	●
Not preserving base class invariants	Structural, Behavioral	Classes, Statements	Maintainability, Reliability	Invariants, assertions, or other information from the base class is not “inherited” in the subclass	●	⭘	⭘	⭘
Converting non-object code straight into objects	Structural	Classes, Methods, Attributes	Maintainability	Classes that look like modules, libraries, data containers, or single functions.	●	⭘	⭘	●
Letting objects become bloated	Structural	Classes, Methods, Attributes	Maintainability	Classes have too many data members and methods or have very large methods.	●	○	⭘	●
Letting objects ooze	Structural	Classes, Visibility	Maintainability	The information gives access on far too many information	●	○	○	●
Creating swiss army knife objects	Historic	Versions, Classes	Maintainability	A class or class hierarchy is changed for different reasons over time (again and again).	●	○	⭘	●
Creating hyperspaghetti objects and subsystems	Structural	Classes, Methods, Calls	Maintainability, Reliability	Classes call many other classes and the coupling between classes is high (and not clearly separated by components such as layers)	●	○	⭘	●
Copying objects	Control	Classes, Statements	Reliability, Maintainability	Objects are copied in a wrong manner (e.g., to swallow, by assignment, slicing, etc.)	⭘	●	●	●
Testing objects for quality and identity	Control	Statements	Reliability, Maintainability	Multiple checks if a object is “null”	⭘	⭘	●	●
Not keeping track of objects	Control	Statements	Reliability, Efficiency, Maintainability	Information (objects) are disposed by one object but not another object.	⭘	⭘	●	●
Consuming memory	Control	Statements	Efficiency	Construction of object with many	●	⭘	⭘	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
inadvertently				large instance variables				
Confusing switch statements and polymorphism	Structure, Control	Statements	Maintainability, Reliability	Switch statements o if-the-else cascades are used to differentiate between object types.	⊙	○	●	○

Another group of pitfalls was collected by Daconta et al. for the Java language. As these pitfalls are platform-specific only an excerpt of the 50 documented pitfalls in (Daconta et al., 2000) and the 50 pitfalls in (Daconta et al., 2003) is given in the following table:

Table 38. *Java Pitfalls by (Daconta et al., 2000) (Daconta et al., 2003)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Hidden Fields	Structure	Classes, Attributes, Inheritance	Maintainability, Reliability	A field in a subclass overrides the field of the same name in a superclass.	●	○	○	⊙
Forward References	Control	Statements	Compilability	Referencing a local variable in the same scope before it is defined.	●	○	⊙	●
Use StringBuffer instead of '+'	Control	Statements	Efficiency	Using concatenation for Strings	○	○	●	●
Too many Submits	Control	HTML, JSP, Statements	Efficiency, Reliability	Duplicated submits as the processing is too slow	○	⊙	●	●

Pitfalls on a similar language-specific level are known for programming languages such as C (Koenig, 1989), more Java (Laffra, 1996), or technologies such as Jarkarta tools (Dudney & Lehr, 2003).

4.18 Principles (Design Principles)

One of the commonly used terms for best practices on the software architecture and design level are "principles". In general, principles are guidelines that should be followed, However, the absence of a principle or their inversion represent problematic (micro-)structures in the software design that have a negative impact on the quality of a software system (i.e., the software design). In the literature they are defined as follows:

- *"[principles] govern the micro-structure of object-oriented software applications" (Martin, 2000)*
- *"design principles can provide us with valuable tips for curing architecture smells." (Roock & Lippert, 2006)*

One of the best known sets of principles was collected by Robert C. Martin. He envisioned principles as the abstract root of more specific heuristics.

Table 39. *Principles collected by (Martin, 2000) and (Roock & Lippert, 2006)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
DRY - Don't Repeat Yourself	Semantic	Methods, Statements	Maintainability, Reliability	Do not write the same or similar code more than once. Also called "Once and Only Once" principle.	●	○	●	○
SCP - Speaking Code Principle	Semantic	Names, Comments	Maintainability	The code should communicate its purpose. Comments in the code could indicate that the code communicates its purpose insufficiently.	●	◎	◎	●
OCP - Open Closed Principle	Structure	Classes, Inheritance	Maintainability	Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.	◎	○	○	●
LSP - Liskov Substitution Principle	Structure	Classes, Inheritance, Attributes, Methods	Maintainability	One instance of a class must be usable for all instances where the type is the superclass. Not only is it required that the compiler translates the source code, but after the modification the system must still function correctly.	●	○	○	●
DIP - Dependency Inversion Principle	Structure	Classes, Inheritance, Calls	Maintainability	High-level concepts shall not depend on low-level concepts/implementations. The dependency should be vice versa, because high-level concepts are less liable to change than low-level concepts. One can introduce additional interfaces to adhere to the principle.	●	○	○	●
ISP - Interface Segregation Principle	Structure	Classes, Inheritance, Calls	Maintainability	Interfaces should be small. The methods of single interfaces should possess a high number of couplings.	●	○	○	●
REP: Reuse/Release Equivalency Principle	Unknown	Classes, Inheritance, Calls	Portability	The elements that are reused are the elements that will be released.	○	○	○	●
CRP: Common Reuse Principle	Unknown	Classes, Inheritance, Calls	Portability	The classes of a package are reused as a whole.	○	○	○	○
CCP: Common Closure Principle	Historic	Versions, Classes	Maintainability, Portability	The classes of a package shall be closed against the same type of changes. If a class must be changed, all classes of the package must be changed as well.	●	○	○	○
ADP: Acyclic Dependencies Principle	Structure	Packages, Calls	Maintainability, Portability	The dependency structure between packages shall be acyclic.	●	○	◎	○
SDP: Stable Dependencies Principle	Structure	Packages, Calls	Maintainability, Portability	A package shall only depend on packages that are at least as stable as itself.	●	○	◎	●
SAP: Stable Abstractions Principle	Historic	Versions, Classes	Maintainability, Portability	The more stable a package is, the more abstract it should be. Instable packages should be concrete.	●	○	○	●
TDA: Tell, Don't Ask	Structure	Calls, Statements	Maintainability, Efficiency	Don't ask an object about an object, but tell it what to do. Similar to the "Law of Demeter": Each object shall only talk to "friends," i.e. only to objects that it retains as fields or receives as parameters.	●	○	●	◎

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
SOC: Separation Of Concerns	Structure	Classes, Methods, Interfaces	Maintainability	Do not mix several concerns within one class.	●	⭘	○	●

Further, more basic principles were described by (Coad & Nicola, 1993) (Appendix C) that represent potential threats to object-oriented source code. However, as many of these principles are not applicable to architecture and design products only an excerpt is listed.

Table 40. *Principles by (Coad & Nicola, 1993)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
The "-er-er" principle	Semantic	Names	Maintainability	Class names that end in "-er" (e.g., Changer, Controller, etc.) do probably not represent real objects.	●	○	⭘	●
The throw out the middle man principle	Structure	Calls, Statements	Maintainability, Efficiency	Throw out objects that do nothing more than take a request and pass it on to another object.	●	○	⭘	⭘
The strip search principle	Semantic	Names	Maintainability	Compound (CamelCase) names should be analyzed for similarity with other names.	●	⭘	⭘	○
The "it's my name; generalize it" principle	Semantic	Names	Maintainability	Class names should be as general as possible (and reasonable)	●	⭘	⭘	○
The "more than just a data hider" principle	Structure	Class, Attributes, Methods	Maintainability	An object acts as just a data hider when another object sends it a message.	●	●	○	●
The "don't butt into someone else's business" principle	Control	Statements	Efficiency, Maintainability	Objects shouldn't send other objects a message to peek at its values and then another to get the work done.	●	○	●	⭘

4.19 Puzzles / Puzzlers

A relatively new concept "puzzle" was used by Bloch and Gafter in their book to describe problems in object-oriented Java systems (Bloch & Gafter, 2005). These puzzles represent platform-specific problems of the software system that appear correct but are unknowingly wrong, complicated, or cumbersome. In general, puzzles are problems that are associated with one or more idiosyncrasies of the (Java) programming language. In the literature they are defined as follows:

- *"Puzzles exploit counterintuitive or obscure behaviors that can lead to bugs" (Bloch & Gafter, 2005)*

As these puzzles are platform-specific only an excerpt of the 95 documented puzzles in (Bloch & Gafter, 2005) (not including the 81 more general traps in appendix A) is given in the following table:

Table 41. *Puzzles by (Bloch & Gafter, 2005)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Puzzle 2: Time for a change	Control	Statements	Reliability, Functionality	Not all decimals can be represented by floating point	○	○	●	●
Puzzle 11: The last Laugh	Control	Statements	Reliability, Functionality	The + operator performs string concatenation if and only if one of its operands is of type String.	○	○	●	●
Puzzle 25: Incle-ment Increment	Control	Statements	Reliability, Functionality	Do not assign the same variable more than once in a single expression.	○	○	●	●
Puzzle 29: Bride of Looper	Control	Statements	Reliability, Functionality	Once a float reaches NaN further computations might get corrupted.	○	○	●	○
Puzzle 47: Well, Dog my Cats!	Control	Statements	Reliability, Functionality	A single copy of each static field is shared among its declaring class and subclasses.	●	○	●	○
Puzzle 59: What's the difference?	Control	Statements	Reliability, Functionality	Integer literals beginning with a "0" are interpreted as octal values.	○	○	●	○

While these are platform-specific problems they are nevertheless relevant to the platform-independent level. As models on the PIM level are going to be transformed to the PSM level these problems should be taken into consideration either while modeling the PIM or in the development of PIM to PSM transformators. Being system-independent the consideration of these problems in general-purpose transformers or quality-checking transformers (i.e., on the PSM level) seems better in order to not overload the PIM level (that should not consider all platform-specific quality defects, e.g., for Ada or Cobol).

4.20 Rules (Design Rules)

The concept "rules" is used by Johnson and Foote to describe problems in object-oriented languages such as Smalltalk. These rules represent guidelines how these systems should be build. In the literature they are defined as follows:

- *"[rules] help the designer create standard protocols, abstract classes, and object-oriented frameworks." (Johnson & Foote, 1988)*
- *"[rules] include both good practices for this kind of design and specific requirements from the stakeholders for this system. "(Liu et al., 2002)*

Johnson & Foote present several design rules for developing better, more reusable object-oriented programs.

Table 42. *Design rules by (Johnson & Foote, 1988)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Recursion Introduction	Structure, Semantic	Classes, Associations, Calls, Names	Maintainability	If one class communicates with a number of other classes, its interface to each of them should be the same (i.e., similar naming of methods).	●	○	○	○
Eliminate Case Analysis	Structure, Control	Statements	Maintainability	It is almost always a mistake to check the class of an object.	●	○	○	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Reduce the Number of Arguments	Structure	Methods	Maintainability, Efficiency	Messages that have a dozen or more arguments are hard to read (except constructors).	●	○	◎	●
Reduce the Size of Methods	Structure	Methods, Statements	Maintainability, Reliability	It is easier to subclass a class with small methods, since its behavior can be changed by redefining a few small methods instead of modifying a few large methods.	●	○	●	●
Hierarchies should be Deep and Narrow	Structure	Classes, Inheritance	Maintainability	A well developed class hierarchy should be several layers deep.	●	○	○	◎
The Top of the Hierarchy should be Abstract	Structure	Classes, Methods, Inheritance	Maintainability	Inheritance for generalization or code sharing usually indicates the need for a new subclass.	●	○	○	●
Minimize accesses to variables.	Structural	Classes	Maintainability	Classes can be made more abstract by eliminating their dependence on their data representation.	●	◎	○	●
Subclasses should be specializations	Structure, Semantic	Classes, Inheritance	Maintainability	Subclass redefines method, adds no new ones, or	●	○	○	●
Split Large Classes	Structure	Classes, Methods, Statements	Maintainability	Large classes should be viewed with suspicion and held to be guilty of poor design until proven innocent.	●	○	○	●
Factor Implementation differences into subcomponents	Structure, Semantic	Classes, Methods, Statements	Maintainability	If some subclasses implement a method one way and others implement it another way then the implementation of that method is independent of the superclass. It is likely that it is not an integral part of the subclasses and should be split off into the class of a component.	●	○	○	●
Separate Methods that do not Communicate	Structure	Classes, Methods, Calls	Maintainability	A class should almost always be split when half of its methods access half of its instance variables and the other half of its methods access the other half of its variables.	●	○	◎	●
Send messages to components instead of to self.	Structure	Classes, Methods, Calls	Maintainability	An inheritance-based framework can be converted into a component-based framework black box structure by replacing overridden methods by message sends to components.	●	○	○	●
Reduce implicit parameter passing.	Structure	Classes, Methods, Attributes	Maintainability	A class is hard to split into two parts because methods that should go in different classes access the same instance variable.	●	○	◎	●

Liu collected several production rules to identify inconsistencies in UML models. However, several rules such as cleanup-rules are not listed due to their specific nature.

Table 43. *Inconsistency Rules by (Liu et al., 2002)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
An object is absent from the specialized sequence diagram.	Structure	Sequence	Maintainability	If feature A is a specialization of feature B illustrated in the corresponding diagrams, then an inconsistency occurs if an object that appears in B's diagram, is absent from that of A.	●	○	⊙	●
Conflicting states reachable in state diagrams.	Structure	State	Reliability	When two features have overlapping specifications, conflicting states may be reached simultaneously.	●	○	⊙	○
No Attributes may have the same name within a Classifier.	Structure	Classes, Attributes	Compilability	Attributes with the same name	●	○	○	○
A design model should obey the Law of Demeter.	Structure	Classes, Attributes	Conformance, Maintainability	When a Singleton pattern is used in a design, no other class objects should keep a reference to the singleton class object. (A Singleton pattern is recognized if the class has a static method returning an instance of the class and a static attribute that stores instances of this class.)	●	○	○	⊙

Furthermore, Liu described further rules in her master thesis. We list the one not described in Table 43.

Table 44. *Inconsistency Rules by (Liu, 2002)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
An object of a behavioral diagram is undefined in class diagrams.	Structure	Classes, Objects, Behav. Diagrams	Compilability	Definition of an object is missing in the class diagrams.	●	○	○	●
A message of a behavioral diagram is undefined in the corresponding class definition.	Structure	Classes, Methods, Behav. Diagrams	Compilability	Definition of a method is missing in the class diagrams.	●	○	⊙	●
A message in a behavioral diagram has a parameter that is absent from its correspondence in the class diagram.	Structure	Classes, Methods, Behav. Diagrams	Compilability	Definition of a parameter is missing in the class diagrams.	●	○	⊙	●
A message in a behavioral diagram is missing a parameter whose correspondence exists in the class diagram.	Structure	Classes, Methods, Behav. Diagrams	Compilability	Definition of a parameter is missing in the class diagrams.	●	○	⊙	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
A message is absent (from the specialized sequence diagram)	Structure	Sequence	Maintainability	If feature A is a specialization of feature B illustrated in the corresponding diagrams, then an inconsistency occurs if a message that appears in B's diagram, is absent from that of A.	●	○	◘	●
The AssociationEnds must have a unique name within the Association.	Structure	Associations	Compilability	Association Ends have the same name.	●	○	○	○
At most one AssociationEnd may be an aggregation or composition.	Structure	Associations	Compilability	Multiple association Ends are aggregation or composition.	●	○	○	○
When multiple classes in a package are accessed from outside the package, a Façade pattern can be used and a Façade class should be placed as a common interface to the package.	Structure	Classes, Calls	Maintainability	Missing Façade class.	●	○	◘	●

4.21 Sins (Code sins)

The concepts (design-oriented) "sins" are typically used as a term to emphasize the problems accompanied with quality defects. Several authors use the term to describe recurring and named problems.

Furthermore, at least in German the concept "code sin" (ger. "Code Sünden") is used as a wrapper for code smells and design flaws (Simon et al., 2006).

The concept of "sins" was used by Howard in the book "19 Deadly Sins of Software Security" (Howard et al., 2005) to describe concrete situations where security holes are opened unknowingly or by lax behaviour of the developers.

Table 45. *Security Sins by (Howard et al., 2005)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Buffer Overrun	Control	Statements	Functionality	A program allows input to write beyond the end of the allocated buffer.	●	●	◘	●
Format String Problems	Control	Statements	Functionality	Input from an untrusted user is allowed to pass through a format String; this can result in anything from arbitrary code execution to spoofing user output.	●	●	◘	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Integer Overflows	Control	Statements	Functionality, Reliability	Integer overflow crashes and logic errors due to failure to check the range on integer types.	◉	○	○	●
SQL Injection	Control	Statements, Queries	Functionality, Reliability	Building SQL statements with input from untrusted or unknown users – i.e., they can "inject" their own commands into the SQL statements.	●	●	◉	●
Command Injection	Control	Statements	Functionality, Reliability	Untrusted user input is passed to a compiler or interpreter, or worse, a command line shell.	●	◉	◉	●
Failing to handle Errors	Control	Statements	Functionality, Reliability	A program's error handling strategy leads to the program crashing, aborting, or restarting – a weakness exploited in denial of service attacks.	●	●	◉	●
Cross-site Scripting	Control	Statements	Functionality, Reliability	Unvalidated input from the user is echoes directly back to the users (e.g., via a web page) – giving it access to anything your website could do, including retrieving cookies, etc.	●	●	◉	●
Failing to protect network traffic	Semantic	Statements	Functionality	Transmitting data over the network, even if that data is not private - attackers can eavesdrop, replay, spoof, etc. any unprotected data sent over the network.	◉	●	◉	●
Use of magic URLs and Hidden Form fields	Control	Statements, HTML	Functionality	Passing sensitive or secure information via the URL query string or hidden HTML form fields.	●	●	◉	●
Improper use of SSL and TLS	Semantic	Statements	Functionality	Using most SSL and TLS APIs without checking for certificates from lax authorities, subtly invalid certificates, or stolen/revoked certificates.	○	●	○	◉
Use of weak password-based systems	Semantic	Statements	Functionality	Using passwords without considering risks such as phishing, social engineering, eavesdropping, keyloggers, brute force attacks, etc..	○	●	○	◉
Failing to store and protect data security	Semantic	Statements	Functionality	Information spends more time stored on disk than in transit – without equivalent permissions and encryption for any data stored.	○	●	○	◉
Information leakage	Semantic	Statements, HTML	Functionality	Giving helpful feedback allows attackers to learning about the internal details the system (e.g., if the password or name was invalid)	○	●	○	◉
Improper File Access	Semantic	Data Access	Functionality	An attacker can slip changes in files from a filesystem (e.g., new file or a link), particularly if the files are accessed over the network.	○	●	○	◉
Trusting network name resolution	Control	DNS Access	Functionality	Domain names on a server or workstation are overridden and subverted with a local HOSTS file.	○	●	○	◉
Race conditions	Control	Statements	Functionality, Reliability	A race condition occurs when two different execution contexts are able to change a resource and interfere with each other.	●	◉	◉	○

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Unauthenticated key exchange	Control	Statements	Functionality	Exchanging a private key without properly authenticating the entity/machine/service that the system is exchanging the key with.	⊙	⊙	⊙	●
Cryptographically strong random numbers	Control	Statements	Functionality	Use of weak (e.g., small) random numbers an attacker can use to breach the security of the system.	⊙	○	⊙	●
Poor Usability	Semantic	Security features	Functionality	Security only works if the secure way also happens to be the easy way.	⊙	○	○	○

4.22 Smells

The concept "smell" or "bad smell" was coined by Kent Beck and Martin Fowler in the Book "Refactoring: Improving the Design of Existing Code" (Fowler, 1999). These smells represent problematic parts of the software system that seem wrong, complicated, or cumbersome to an experienced developer. In general, smells are problems that are associated with one or more specific refactorings (i.e., concrete treatments) that might be applied to remove the smells. In the literature they are defined as follows:

- *"[Smells] … suggest (sometimes they scream for) the possibility of refactoring"* (Fowler, 1999)
- *"Smells (especially code smells) are warning signs about potential problems in code. Not all smells indicate a problem, but most are worthy of a look and decision."* (Wake, 2003)
- *"[Smells] are present when the existing system structure hampers or even prevents modifications."* (Roock & Lippert, 2006)
- *"Code smell is a popular expression among Extreme Programming practitioners corresponding to signs that suggest that some parts of the code are problematic or violate programming guidelines."* (Correa & Werner, 2004)
- *"A common category of problem in your code that indicates the need to refactor it"* (Ambler & Sadalage, 2006)
- *"… code smell is any symptom that indicates something may be wrong. It generally indicates that the code should be refactored or the overall design should be reexamined."* (Wikipedia, http://en.wikipedia.org/wiki/Code_smell)
- *"A code smell is a hint that something has gone wrong somewhere in your code. Use the smell to track down the problem."* (WikiWikiWeb, http://c2.com/cgi/wiki?CodeSmell)

Beside the smells on the code or design levels many other problems were described using the smell metaphor. Today, we have smells on different abstraction layers, for development phases, or technologies such as architecture smells (Roock & Lippert, 2006), test smells (Deursen et al., 2001), aspect smells (Monteiro & Fernandes, 2006), database smells (Ambler & Sadalage, 2006), OCL smells (Correa & Werner, 2004), projects smells (Elssamadisy & Schalliol, 2002), or user story smells (Cohn, 2004).

In the following sections we will list most of these smells that were found in the literature survey. The first large collection of smells were collected by Kent Beck and Martin Fowler (Fowler, 1999).

Table 46. *Bad smells in code (Fowler, 1999)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Alternative classes with different interfaces	Semantic, Control	Classes, Methods, Statements, Names	Maintainability	Two classes are doing similar thing and have similar interfaces but use different method names, method structures, or are not related (i.e., are not using a shared superclass or interface).	●	○	◎	○
Comments	Semantic	Notes / Comments	Maintainability	Superfluous or redundant description of the software.	●	○	◎	●
Data class	Structure	Class, Attributes, Methods	Maintainability	Classes that do almost exclusively store information for other classes. Optionally, these classes have getter and setter methods for the attributes.	●	●	○	●
Data clumps	Structure, Message	Class, Attributes, Parameters, Local Variables	Maintainability	Data items (i.e., attributes, parameters, local variables, etc.) that appear in groups all over the system (e.g., id, surename, forename, salary).	●	●	◎	○
Divergent change	Historic	Versions, Classes	Maintainability	A class is changed for different reasons over time.	●	○	◎	●
Duplicated code	Semantic	Methods, Statements	Maintainability, Reliability	Identical code passages are distributed over the whole system	●	○	●	○
Feature envy	Structure	Classes, Calls	Maintainability	A method is occupied more with data and methods in other classes than its own.	●	○	◎	◎
Inappropriate Intimacy	Structure	Classes, Attributes, Calls	Maintainability	Classes access far too many internal parts (attributes or methods) of other classes.	●	○	◎	◎
Incomplete library class	Structure	Classes	Maintainability, Portability	An external (library) class that misses some functionality but cannot be changed)	●	○	●	●
Large class	Structure	Classes, Methods, Statements	Maintainability, Portability	A class with far too many methods, attributes, and consequently responsibilities.	●	○	○	●
Lazy class	Structure	Classes, Methods, Statements	Maintainability	A class that isn't doing much.	●	○	◎	●
Long method	Structure	Methods, Statements	Maintainability, Reliability	A very large method.	●	○	●	●
Long parameter list	Structure	Methods	Maintainability, Efficiency	A method with too many parameters.	●	○	◎	●
Message chains	Structure	Calls, Statements	Maintainability, Efficiency, Portability	One object asks another object for data in a third object (and so on).	●	○	◎	◎
Middle man	Structure	Calls, Statements	Maintainability, Efficiency	A method delegates the functionality to another method (or class).	●	○	◎	◎
Parallel inheritance hierarchies	Historic, Structure	Classes, Inheritance	Maintainability	Creating a subclass in one hierarchy requires the creation of another subclass in a different hierarchy	●	○	○	○
Primitive obsession	Structure	Attributes, Parameters, Local Variables	Maintainability	Far too many primitive types are used (in a class or method)	◎	○	●	●

Refused bequest	Structure	Classes, Attributes, Methods	Maintainability	Subclasses that inherit attributes and methods that they do not use.	●	○	⊙	⊙
Shotgun surgery	Historic	Versions, Classes	Maintainability, Portability	Several classes are changed in a group every time a specific kind of change is to be made.	●	○	○	○
Speculative generality	Structure	Relations, Calls, Inheritance	Maintainability, Portability, Reliability	Classes, methods, attributes, or code passages do only exist for future, potential features	●	○	⊙	●
Switch statements	Structure, Control	Statements	Maintainability, Reliability	Similar Switch statements are used to differentiate between behavior in different classes.	⊙	○	●	○
Temporary field	Semantic, Control	Statements	Maintainability, Reliability	Fields are only set at specific times and the time when the content is valid is nondeterministic.	●	○	●	●

Table 47. *Code smells by (Wake, 2003)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Type Embedded in Name	Semantic	Names	Maintainability	Type information is redundantly encoded in the name / identifier of an attribute, method, etc.	●	⊙	⊙	●
Uncommunicative Name	Semantic	Names	Maintainability	The name does not communicate the intent (e.g., short names, abbreviations, …).	●	⊙	⊙	●
Inconsistent Names	Semantic	Names	Maintainability	Names are not consistent throughout the system.	●	⊙	⊙	○
Complicated Boolean Expression	Structure	Statements	Maintainability	Complex condition involving Boolean operators (“and”, “or”, “not”).	●	⊙	●	●
Magic Numbers	Control	Statements	Maintainability, Reliability	Constants that appear multiple times in the system	●	●	●	○
Null Check	Control	Statements	Reliability	Multiple checks if a object is “null”	⊙	⊙	●	●
Special Case	Control	Statements	Maintainability, Reliability	Check for particular values or states before doing work	●	⊙	●	●
Simulated Inheritance (Switch Statement)	Structure, Control	Statements	Maintainability, Reliability	Similar Switch statements are used to differentiate between behavior in different classes.	⊙	○	●	○

Table 48. *Code smells by (Kerievsky, 2005)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Indecent Exposure (aka: Unnecessary Openness)	Structural	Classes	Maintainability	The classes (or packages, etc.) gives access on far too many information	●	⊙	○	●
Solution Sprawl	Semantic	Classes, Statements	Maintainability	Many small features are realized without consolidating or using existing features (i.e., methods)	●	○	⊙	●

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Combinatorial Explosion	Structural	Classes, Methods, Statements	Maintainability	Code duplication for many slightly different features (e.g., queries)	●	⭘	●	○
Oddball Solution	Semantic	Classes, Names, Statements	Maintainability	Different solutions do exist for the same problem.	●	⭘	●	○
Conditional Complexity	Control	Statements	Maintainability	Large and complex conditional statements (i.e., if, switch etc.)	●	⭘	●	●

Table 49. *Code smells by (Tourwé & Mens, 2003)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Obsolete Parameter	Structural, Behavior	Classes, Methods, Parameters	Maintainability, Reliability	Parameter are not used in any of implementations of the given class	●	⭘	●	●
Inappropriate Interfaces	Structural	Classes, Methods	Maintainability, Reliability	Differences between common interfaces of direct subclasses and the interface of the root class.	●	○	○	⭘

Table 50. *Architecture smells (Roock & Lippert, 2006)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Obsolete Classes, Unused Element (Class)	Structure	Classes, Calls	Maintainability	Classes are not in the control path and not used in the system.	●	○	⭘	●
Treelike Dependency Hierarchies	Structure	Classes, Calls	Maintainability	Classes are only used by one other class.	●	○	⭘	⭘
Static Cycles (Class)	Structure	Classes, Calls	Maintainability, Portability	Classes are used in a cycle	●	○	⭘	○
Visibility of Dependency Graph	Structure	Classes, Calls	Maintainability, Portability	Internal information of classes is used by other classes	●	○	●	○
Type Queries	Structure, Control	Classes, Statements	Maintainability, Reliability	The type of an object is identified programmatically.	⭘	○	●	○
List-like Inheritance Hierarchies	Structure	Classes, Inheritance	Maintainability	Classes have only one subclass	●	○	○	●
Subclasses do not redefine methods	Structure	Classes, Inheritance	Maintainability	Subclasses redefine no methods of the superclass	●	○	●	●
Hierarchy without polymorphy	Structure	Classes, Inheritance	Maintainability	Inheritance hierarchies without polymorphy. Superclasses that are used only sparely or never.	●	○	○	⭘
Parallel Inheritance Hierarchies	Historic, Structure	Classes, Inheritance	Maintainability	Creating a subclass in one hierarchy requires the creation of another subclass in a different hierarchy	●	○	○	○

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Deep inheritance Hierarchy	Structure	Classes, Inheritance	Maintainability	The inheritance tree is too depth.	●	○	○	◎
Unused Elements (Package)	Structure	Packages, Calls	Maintainability	Packages are not in the control path and not used in the system.	●	○	◎	●
Static Cycles (Package)	Structure	Packages, Calls	Maintainability, Portability	Packages are used in a cycle	●	○	◎	○
Package too small	Structure	Package	Maintainability	A package with far too few classes (or other types – e.g., enums) and consequently responsibilities.	●	○	○	●
Package too large	Structure	Package	Maintainability	A package with far too many classes (or other types – e.g., enums) and consequently responsibilities.	●	○	○	●
Deep or Unbalanced Package Hierarchy	Structure	Packages, Inheritance	Maintainability	A package hierarchy that is too deep or unbalanced.	●	○	○	◎
Packages Not Clearly Named	Semantic	Package Namess	Maintainability, Reliability	Package names that occur multiple times in the system or that does not communicate its intention.	●	○	◎	●
No Subsystems	Structure	Subsystem	Maintainability	No subsystems defined	●	○	○	●
Subsystem too large	Structure	Subsystem	Maintainability	Subsystem with far too many packages.	●	○	○	●
Subsystem too small	Structure	Subsystem	Maintainability	Subsystem with far too few packages.	●	○	○	●
Too many Subsystems	Structure	Subsystems	Maintainability	Too many subsystems defined.	●	○	○	●
Subsystem-API Bypassed	Structure	Subsystem, Calls	Maintainability, Reliability	Clients are bypassing the subsystem-API.	●	○	●	●
Subsystem-API too Large	Structure	Subsystem	Maintainability	Subsystem-API with far too many open packages.	●	○	○	●
Static Cycles (Subsystem)	Structure	Subsystems, Calls	Maintainability, Portability	Subsystems are used in a cycle	●	○	●	◎
Overgeneralization	Structure	Subsystems, Calls	Maintainability, Reliability	Clients need to reimplement many “real” / non-abstract functionality.	●	○	●	◎
No Layers	Structure	Layers	Maintainability	No layers defined	●	○	○	●
Upward references between Layers	Structure	Layers, Calls	Maintainability, Portability	Lower-level layers use upper-level layers	●	○	◎	●
Strict Layers Violated	Structure	Layers, Calls	Maintainability, Portability	Upper-level layer skipped middle-level and used lower-level layer.	●	○	◎	●
Inheritance between protocol-oriented Layers	Structure	Layers, Inheritance	Maintainability, Portability	Classes in layers inherit from another or have a common superclass.	●	○	○	◎
Too many Layers	Structure	Layers	Maintainability	Too many layers defined.	●	○	○	●
References between Vertically Separated Layers	Structure	Layers, Calls	Maintainability, Portability	Layers use sister-layers.	●	○	◎	●

Table 51. *OCL smells by (Correa & Werner, 2004)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Magic Literal	Control	OCL code	Maintainability	Numeric or string literal that appears in the middle of an OCL expression without explanation.	●	⭘	●	●
And Chain	Control	OCL code	Maintainability	Complex Boolean expression (esp. AND)	●	○	●	●
Long Journey	Structure, Control	OCL code	Maintainability	An OCL expression that traverses many associations between different classes of the model.	●	○	●	⭘
Rules Exposure	Structure, Control	OCL code	Maintainability	Business rules details are specified in the pre- or postconditions of system-level operations.	●	○	●	●
Duplicated Code	Structure, Control	OCL code	Maintainability	Duplicated OCL expressions.	●	○	●	⭘

Table 52. *Database smells by (Ambler & Sadalage, 2006)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
Multi-purpose column	Data structure	DB Column, (Attribute)	Reliability, Maintainability	A column is used for several purposes. (Similarily, an attribute in a data class.)	⭘	●	⭘	●
Multi-purpose table	Data structure	DB Table, (Class, Attributes)	Reliability, Maintainability	A table is used to store several types of entities. (Similarily, a class is used to store different types of objects)	⭘	●	⭘	●
Redundant data	Data structure	Database, (Classes, Attributes)	Reliability, Maintainability	Data is stored in different places (e.g., birthday).	⭘	●	⭘	⭘
Tables with too many columns	Data structure	DB Table, Class, Attributes	Maintainability	Items consist of too much data and table probably lacks cohesion.	⭘	●	○	●
Tables with too many rows	Data	DB Data, Instances	Efficiency	Table encompasses too much information.	⭘	●	○	●
"Smart" columns	Data structure	DB Data	Reliability, Maintainability	Data in Columns is encoded – e.g., data type is embedded in a number.	⭘	●	⭘	●
Fear of change	Mental	Database	Maintainability	Afraid to change the database.	○	●	○	○

4.23 Styles, Conventions, and Rules

Finally, another set of concepts that is associated with quality defects are styles, conventions, or rules for source code or software models. Behind every of these conventions or guidelines stands a reasonable rationale or a typical recurring problem in a software system. However, they are mostly used to check or improve the inner "structure" of methods while antipatterns and smells are more concerned with the structure of the software design expressed in classes, packages, or layers. Typically, these styles are targeted to improve or assure the readability and maintainability of a software system. In the literature they are defined as follows:

- *"[conventions are] guidelines for creating effective UML diagrams … [and] are based on proven principles that will lead to diagrams that are easier to understand and work with." (Ambler, 2006)*

In the following sections we will list important collections of these styles that were found in the literature survey. The first larger collection of (bad) styles relevant to software design was collected by Ambler. While most of the 300 styles are applicable to MDSD we will only list an excerpt of the styles.

Table 53. *Style conventions by (Ambler, 2006)*

Name	Type of Quality Defect	Design Entities involved	Quality Aspects affected	Description	PIM level	Domain	Behavior	Local
9. Minimize the number of bubble types	Structure	Diagrams	Maintainability	A diagram that holds more than six elements (bubbles).	●	○	○	●
10. Include White Space in a diagram	Layout	Diagrams	Maintainability	Elements in a diagram that are too close together	●	○	○	●
12. Avoid many close lines	Layout	Diagrams	Maintainability	Several lines close together are hard to follow.	●	○	○	●
16. Reorganize large Diagrams into several smaller ones	Structure	Diagrams	Maintainability	Diagram is too large	●	○	○	●
23. Name common Elements consistently across diagrams	Semantic	Diagrams	Maintainability	One modeling element appears under different names	●	○	○	○
26. Apply color or different fonts sparingly	Layout	Diagrams	Maintainability	More than six colors in a single diagram	●	○	○	●
27. Describe diagrams with notes	Structure	Diagrams	Maintainability	Missing comments / notes about the diagram	●	○	○	●
35. Prefer Naming conventions over Stereotypes	Semantic, Structure	Names	Maintainability	A stereotype such as <<getter>> was used instead of naming the method appropriately (e.g., "get…")	●	○	○	●
58. Begin Use-case names with a strong verb	Semantic	Names	Maintainability	A use case that begins with no or a weak verb (i.e., too general such as "process")	●	○	○	●
63. Name actors with singular domain-relevant nouns	Semantic	Names	Maintainability	A name should accurately reflect its role within your model.	●	○	○	●
76. Avoid more than two levels of use case Associations	Structure	Diagrams	Maintainability	Too many associations (e.g., includes) for a use case.	●	○	○	●
80. Place the inheriting use case below the base use case	Layout	Diagrams	Maintainability	Inheritance order should flow from top to bottom	●	○	○	●
96. Prefer complete singular nouns for class names	Semantic	Classes, Names	Maintainability	Names should not contain abbreviations or verbs and should be in singular form.	●	○	○	●
97. Name Operations with strong verbs	Semantic	Classes, Names	Maintainability	An operation that begins with no or a weak verb (i.e., too general such as "process")	●	○	○	●
112. Model relationships horizontally	Layout	Relationship	Maintainability	Relations with the exception of inheritance should be associated horizontally.	●	○	○	●
114. Model Collabo-	Structural	Collaboration	Maintainability	Missing relationship that describes	●	○	○	●

ration between two elements only when the have a relation-ship				the collaboration				
125. Name unidirec-tional Associations in the same direction	Semantic	Association	Maintainability	An association from A to B where the name implies the other direction (e.g., Item -usedBy-> List should be Item <-uses- List)	●	○	○	●
137. A subclass should inherit every-thing	Structure	Classes	Maintainability	Subclasses that reject attributes or methods from their parents.	●	○	◎	◎
154. Make packages cohesive	Structure	Packages, Calls	Maintainability	Anything within a package should make sense when considered with the rest of the package contents.	●	○	◎	●
156. Avoid cyclic dependencies be-tween packages	Structure	Packages, Calls	Maintainability, Portability	Packages are used in a cycle	●	○	◎	○
158. Strive for left to right ordering of messages	Layout	Diagrams	Maintainability	Message flow that is unordered and makes a zig-zag.	●	○	○	●
213. Name transition events in past tense	Semantic	Names	Maintainability	Names of results of events are al-ready processed.	●	○	○	●
224. Apply connec-tors to avoid un-wieldy activity edges	Layout, Structure	Diagrams	Maintainability	Diagrams with too many lines can be simplified using connectors.	●	○	○	●
248. Have fewer than five swim lanes	Structure	Diagrams	Maintainability	Too many activity partitions.	●	○	○	●
287. Indicate attribute values to clarify instances	Semantic	Objects	Maintainability	Objects are not easily discriminable and Attributes are uncommunicative.	●	○	○	●

Other style conventions for programming languages such as C++ (Sutter & Alexandrescu, 2004), C# (Baldwin et al., 2006) or Java (Vermeulen et al., 2000) are very similar and mostly platform-specific.

Additionally, tools such as Findbugs, PMD, Checkstyle, etc. have large collections of styles they are checking. However, as these are not in the main focus of the VIDE project we will not list them.

5 Domain-specific Quality Defects

This section addresses Task 4.3 ***"Modelling domain-specific parts of the models"*** that was concerned with the identification and formalization of quality defects specific to our particular domain of business applications. The domain-specific variabilities of the domain in respect to quality were analysed and results were used in an extension (i.e., variant) of the core defect model for our specific domain – summarizing and characterizing quality defects of this domain."

Modelling addresses arbitrary applications domains however it is often important to apply the generic modelling concepts to specific application domains to make the models more concrete and understandable to the domain users. The VIDE project therefore focuses on the "particular domain of business applications" to produce programming semantics to enable (programming) behaviour for business oriented personal as described in Deliverable 1 (Vide, 2007a).

This focus on the domain of business applications should also be reflected by Quality Assurance aspects on model level and related methods for quality defect detection that should be focused especially on those aspects important for business applications. Therefore this chapter describes the business application domain and it is specific requirements towards quality assurance.

5.1 Business Application / Business Domain

Business applications in general are software applications to effectively plan, manage a business and its process. Typical examples of business applications are

- **E**nterprise **R**esource **P**lanning (**ERP**) is a unified, integrated business management system to effectively plan and manage organization including data and processes.
- **P**roduct **L**ife cycle **M**anagement (**PLM**) that manages the entire life cycle of a product from its concepts, design and manufacturing, to service and product disposal.
- **C**ustomer **R**elationship **M**anagement (**CRM**) offering structured interaction with customers and which will be used as example in section 5.1.2

A key ingredient of most enterprise systems that execute business applications is a unified database to store data for the various system modules. Therefore most architectures support a Three Tier Architecture that separates the User Interface, Application / Business Logic and the persistence/data layer. Due to customer demands for more flexibility of business processes and cross organizational cooperation SAP evolved this architecture into the E-SOA Architecture.

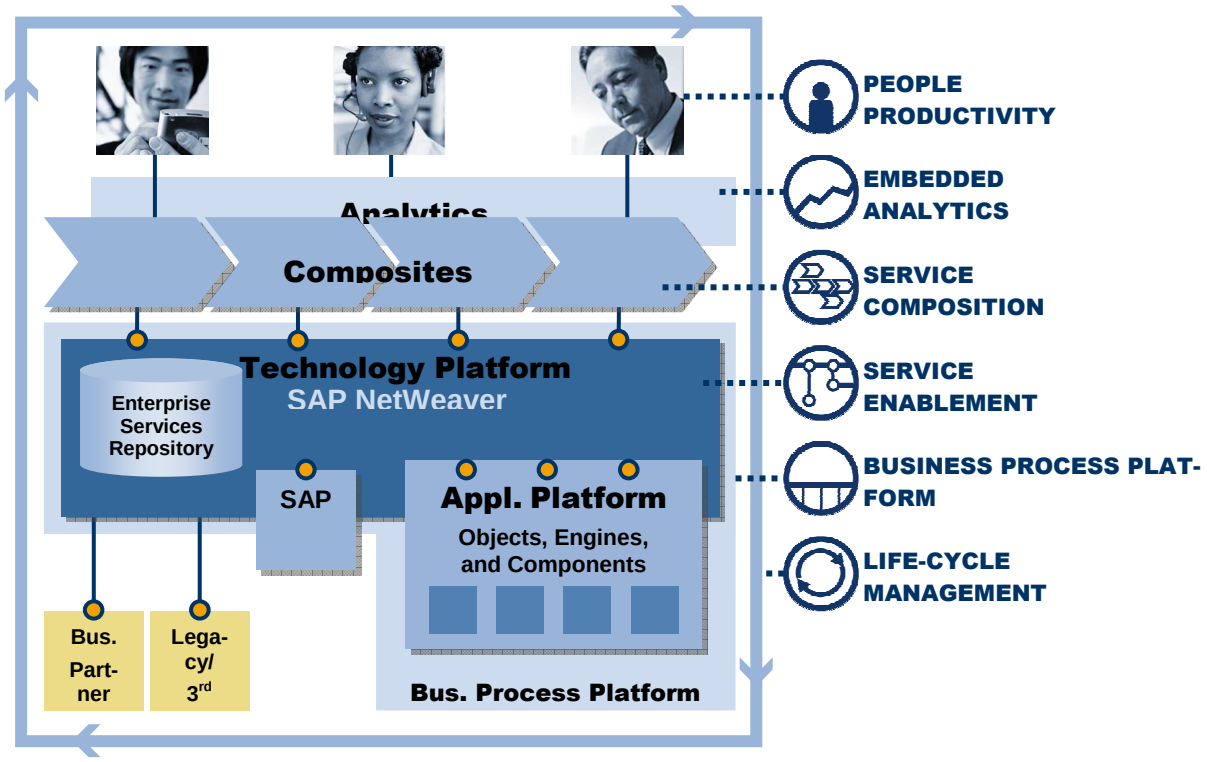

Figure 3. *SAP E-SOA Architecture*

The SAP E-SOA concept enhances of what is considered in the market as a service-oriented architecture (SOA). E-SOA is built upon the technology platform SAP NetWeaver which is evolving into a complete Business Process Platform (BPP) comprising fundamental end-to-end business processes as well as a strong technical infrastructure.

The scope of E-SOA can roughly be characterized by the following six key elements

- **People Productivity**: Pattern-based user interface with role-oriented, consistent portal navigation, cross-application work centers, team collaboration, self services, and integrated office functionality to empower end users to do the best job possible.
- **Analytics**: Seamless integration of transactional and analytical content together with a unified modelling environment for business experts and developers.
- **Service Composition**: Model-driven composition of new services as well as orchestration of existing services to form new business processes and composite applications in order to easily innovate systems as required by changing business processes.
- **Service Enablement**: One common, standard-based service infrastructure with one central Enterprise Service Repository (ESR) to guarantee uniform service definition, implementation, and usage across all types of services (User Interface, cross-application communication) and for all relevant interaction models (synchronous, asynchronous).
- **Business Process Platform**: One BPP shared across all applications provides re-usable business functionality (provided by platform process components) as well as the complete technical infrastructure necessary for e.g., service enabling, re-use, and business process composition.
- **Lifecycle Management**: One common application life cycle management cross all SAP solutions from installation and configuration to operation, change management, and support as a key prerequisite to lowering TCO.

With these key elements, E-SOA defines a complete infrastructure for building and operating the next generation of service-oriented business applications.

The User interface layer (*People Productivity* & *Analytics*) and persistence layer (within *Application Plattform* and build upon the *NetWeaver* stack) are not the main focus for the VIDE project. This analysis therefore focuses on (business relevant) **programming and query aspects** of the middle / application layer (mainly *Objects* within the *Application Platform* and the *Service Composition* layer) with special emphasis on data manipulation through queries due to the often data intense nature in many business application. New architectures such as Service Oriented Architectures (SOA) or SAP e-SOA (TopCased, 2007) have similar abstraction layers.

Data Intense Applications in the context of VIDE means that VIDE application are expected to be data centric and build on top of a persistence layer, such as a for instance an OO database. Data Intense Applications implement a certain characteristics that potentially influence the specific defects of the domain. However some of the characteristics described are of generic nature and not specific to database applications.

5.1.1 Applications for SME

Enterprise/business applications for SME differs from the enterprise applications for larger customers. While SME customers usually implement the same type of operations and processes as bigger customers; they are usually much more diverse in their business operations and process. Therefore the standard business applications need to be much more adaptive to the specific needs. Since SMEs are much more costs sensitive **adaptations**/customizations need a very **efficient** implementation. For the same reasons the Total Cost of Ownership (TCO) is very crucial for SMEs.

5.1.2 CRM example

5.1.2.1 CRM & CRM System

Customer **R**elationship **M**anagement is a management concept, which intends to systematize and improve the relationships between corporations and their customers. It can be defined as a customer-oriented corporate strategy that utilizes modern information and communication technologies to establish long-term, profitable customer relationships through holistic and individual marketing, sales and service instruments (Hippner & Wilde, 2002).

A driving force behind CRM is the awareness that retaining existing customers is significantly cheaper and more profitable than acquiring new customers; whereby customer loyalty is highly correlated with customers' satisfaction with previously bought products and services (Heskett et al., 1994; Hippner et al., 2006). A main objective of CRM is thus to establish deep relationships with customers and to extend them systematically.

To support customer relationship management **CRM systems** provide comprehensive IT solutions. It typically includes interfaces to other corporate information systems such as enterprise resource planning (ERP) or supply chain management (SCM) (Hippner et al., 2004). The actual realisation of a CRM system is vendor specific and depends on the architecture of the overall business software solution. CRM systems may be classified into two distinct functional categories (Hippner et al., 2004).

- **Operational CRM**
  Operational CRM supports marketing, sales and service processes by providing the appli-

cations and tools for supporting direct customer interactions. These operational systems are responsible for controlling and coordinating activities across different customer interaction points (e.g. field service, branch office, campaign, website) and communication channels (e.g. e-mail, phone, personal contact). The avoidance of simultaneous, uncoordinated customer contacts over multiple channels is an exemplary responsibility in this task area.

- **Analytical CRM**
  Analytical CRM systems are concerned with the collection, storage, and analysis of customer data by using business intelligence techniques. Analytical CRM systematically stores all relevant data about customer contacts and reactions (e.g. purchase data, billing and payment, campaign responses, survey responses, returns) in a data warehouse. This data may be combined with demographics and other external data before it is analysed by employing data mining methods or used for answering on-line analytical processing (OLAP) queries.

5.1.2.2 SAP CRM

SAP offers several CRM products such as mySAP CRM, which was recently renamed to SAP CRM (Buck-Emden & Zencke, 2004; SAP, 2007b). This application supports the entire operational CRM field and provides components and functionalities supporting the three fundamental CRM processes marketing, sales, and service. This application is implemented using object-oriented programming and some important business objects of each process are grouped together below (Stürmer, 2006):

- Marketing: Lead
- **Sales**: Opportunity, Customer Quote, Sales Contract, Service Contract, Sales Order, and Service Order
- **Service**: Customer Return, Service Request, and Service Confirmation

5.1.3 Lead and Opportunity Management

Figure 4 shows a Sales Scenario example that focuses on sales processes of enterprises selling one or more products. This involves different things, ranging from Opportunity Management to quotations to customers, sales orders and invoice processing. This figure shows also the different user roles that are involved in each step in the sales process.

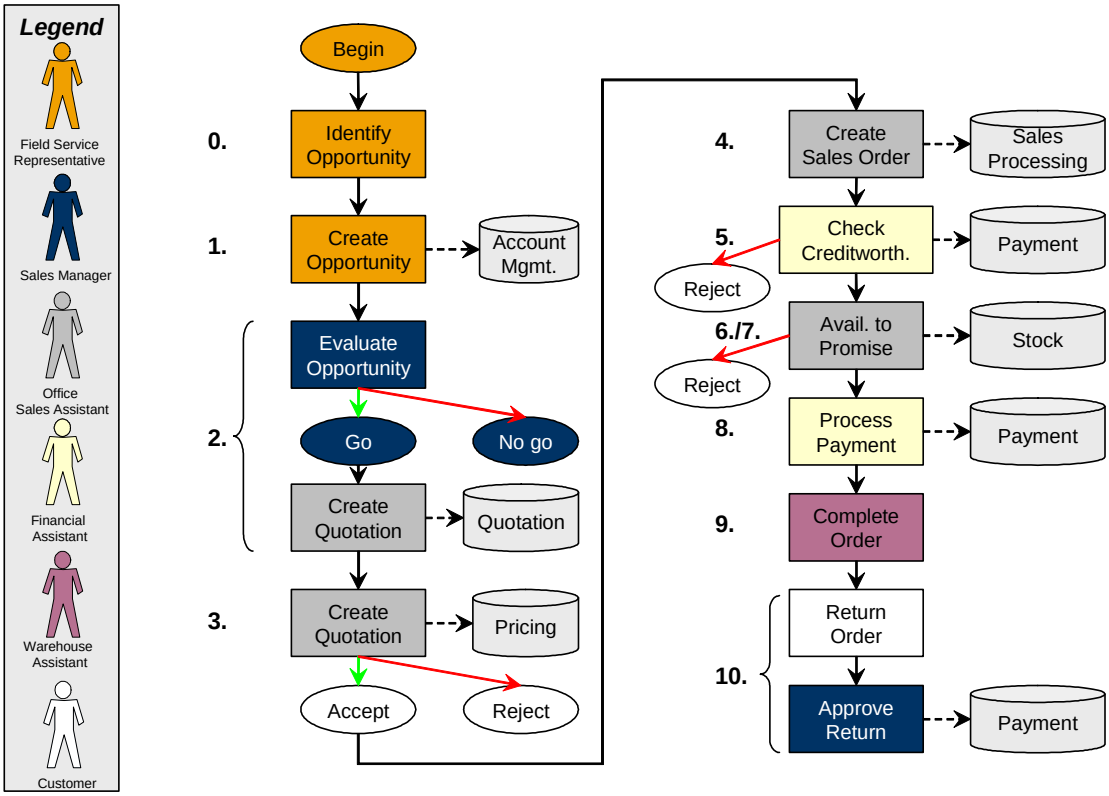

Figure 4. *Sales Scenario*

In the following, we will focus on pre-sales processes such as lead management and opportunity management. These processes support sales personnel in actively tracking potential selling possibilities.

Lead and Opportunity management provides a structured approach to turning an initial recognition of a selling opportunity (i.e., a potential possibility for selling products to a customer) into a sales contract. In that process, the SAP CRM software guides the sales representative through a multilevel process and generates next steps and activity suggestions on the basis of best-practice sales strategies.

The opportunity management process may start with an anonymous address and, by degrees, track additional prospect attributes such as product interests, discretionary budget amounts, likely competitors, and the success probability. Completeness and consistency checks ensure the correctness of the collected data after each step. The accurately documented process improves reporting capabilities: Sales managers can measure their salesperson productivity, campaign effectiveness and can, for example, determine in which sales phases the most prospects were lost (Amberg & Schumacher, 2002; Hippner et al., 2006).

Figure 4 shows also the different steps of the opportunity process. This process starts by the identification and the creation of an opportunity, e.g., after a sales contact at a fair. Then, the opportunity is evaluated and qualified, i.e., feasibility is clarified, information is gathered about the customer, and a selling team is defined. If a go decision is made, a quotation is made and sent to the customer, which either accepts the sales offer or rejects it. After that the opportunity should be closed and the reasons for success or failure should be documented. In the success case, the opportunity becomes a sales order.

In the following, we present in more details some business objects in opportunity management. These objects are shown in Figure 5 and they are discussed briefly below:

- The ***Opportunity*** class uniquely identifies the opportunity and specifies the various involved parties. It holds references to other classes with additional business information and to the documents and activities created during opportunity processing. Some direct attributes of the opportunity class are:
  - priority: specifies the priority of the opportunity.
  - processStatusValidSinceDate: the date when the opportunity entered the current life cycle phase.

- The ***Party*** class represents individuals or organizations involved with the opportunity. Specialized classes may represent customers, suppliers, or employees. Parties are used within the opportunity to specify the prospect, potential competitors, the responsible sales team, and other internal or external stakeholders. Some attributes of a party are:
  - partyType: specifies whether a party is an organization, a business partner, or any specialization of these party types.
  - partyRole/PartyRoleCategory: describe the role of a party in an opportunity.

- The ***SalesForecast*** class contains estimations for the anticipated sale that an opportunity represents. it contains various fields such as
  - expectedRevenueAmount: the expected amount of the opportunity
  - probability: the success probability of the opportunity, expressed in percentage.

- The class ***Item*** represents a product or service which will possibly be sold to the prospect of the opportunity. It contains product information, quantities, and values. An item may be associated with master data product information.

- An opportunity passes through several phases during its lifetime. The class ***SalesCycle*** specifies the sales cycle and the current phase of an opportunity. Other attributes of this class are:
  - salesCycleCode: the sales cycle in which the opportunity exists.
  - phaseProcessingPeriod: the time period for which an opportunity exists in the current phase.

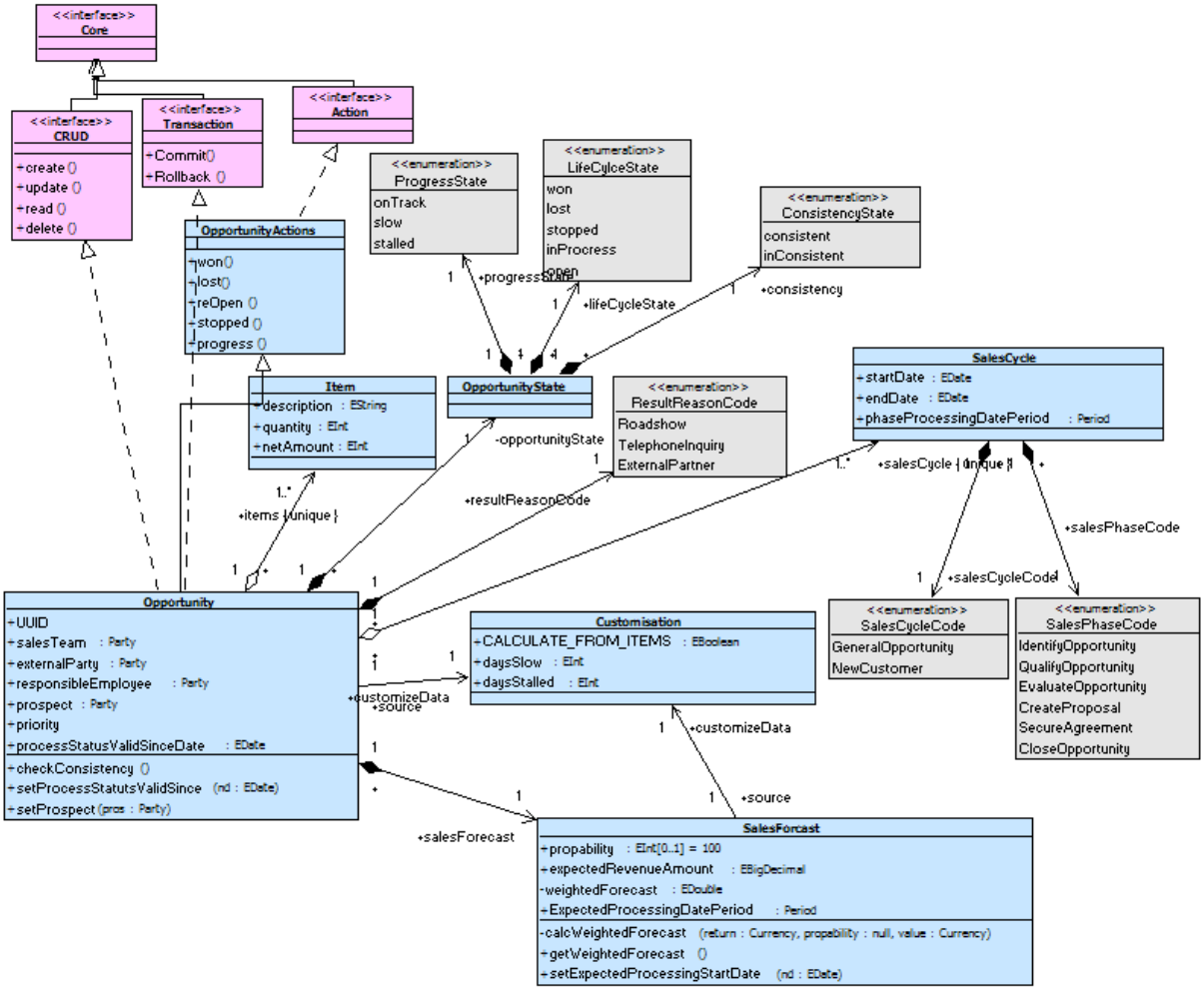

Figure 5. *Main Classes in Opportunity Management*

The Opportunity example described in this section is used as a domain model for the Opportunity Management scenario as described above. The example is a typical structure and behavioural model containing information and business logic that can be found in CRM implementations. The models shown are implemented as EMF models and will be used as basis for defect detection. There faults may be manually introduced into the models to verify the correctness of the error detection.

Figure 6 shows the body of the method setProcessStatusValidSince, which is defined in the class SalesForecast. as an example for a behavioral model for quality detection. The model was created using the tool TopCased (TopCased, 2007). The implemented behavior is also outlined in a Java notation on top of Figure 6.

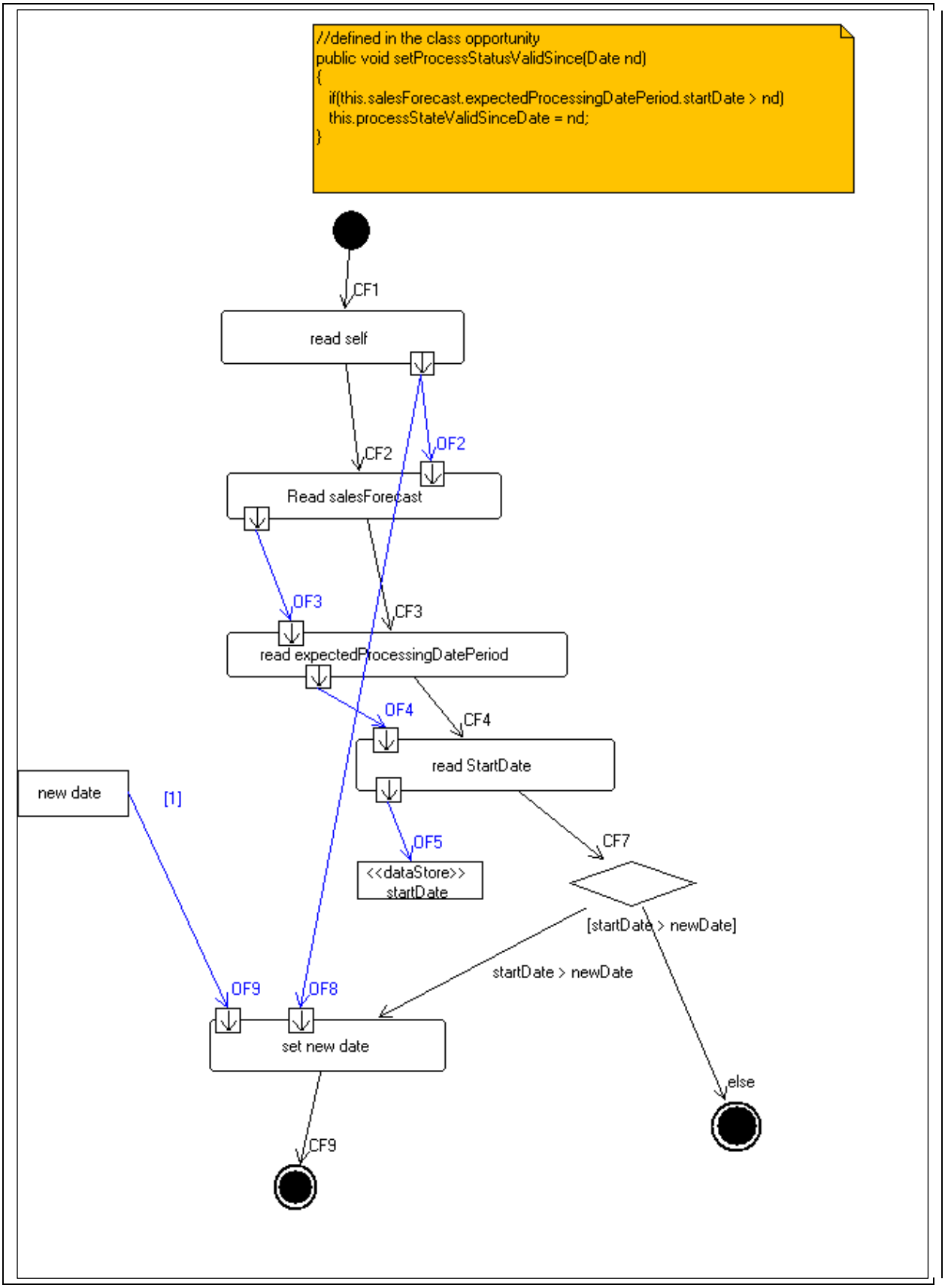

Figure 6. *Diagram of setProcessStatusValidSince()*

The implementation consists of simple decision action (*if*) with a Boolean expression and an assignment.

5.2 Consequences for Quality Assurance in MDSD for the Business Domain

Based on the previous description of the domain and the scenario we conclude that the following characteristics are very important aspects of software systems in the business domain:

- Systems are very large, complex, and incorporate many different application domains (e.g., CRM, Logistics, etc.)
- Development is conducted by large, probably (globally) distributed teams
- Internationalisation such as multiple language support and adoption to country specific regulations, such as taxes
- Evolution is triggered by external factors (e.g., changes of laws or taxes)
- Application are targeted to support larger organizations

Due to the mission critical nature of business applications some quantity characteristics from ISO 9126 (ISO/IEC, 2000a, 2000b) are more crucial and should, therefore, be emphasised when evaluating the behavioural models. The characteristics considered more important are emphasised bold in detailed quantity characteristics descriptions below. For those marked important a short description is given why they are considered important.

5.2.1 Maintainability

The set of attributes that focus on the effort needed to make corrective, preventive, perfective, or adaptive modifications to the software system.

- **Changeability:** Attributes of software that bear on the effort needed for modification, fault removal or for environmental change. (ISO 9126: 1991, A.2.5.2). Business Applications mainly evolve based on changes in the organisations they are supporting as well as requirements from outside of such organisation, such as law, tax and compliance rules. In addition the applications need to be adapted for specific industries (e.g. SAP currently supports 25 industry solutions), countries (e.g. SAP currently supports 120 countries) and languages (e.g. SAP currently supports 31 languages) (SAP, 2007a). Therefore implementation should be easily changeable.
- Analyzability: Attributes of software that bear on the effort needed for diagnosis of functional deficiencies or causes of failures, or for identification of parts to be modified. (ISO 9126: 1991, A.2.5.1).
- Testability: Attributes of software that bear on the effort needed for validating the modified software. (ISO 9126: 1991, A.2.5.4).
- **Stability:** Attributes of software that bear on the risk of unexpected effect of modifications. (ISO 9126: 1991, A.2.5.3). Business applications require a high level of stability since many developers, consultants, etc. work on and change the system – a system a company's main business processes may depend on.
- **Encapsulation/ Modularization:** Business application are targeted to support larger organizations, are often very complex, and cannot be implemented by a small team. Therefore, large development teams and consequently the distribution of work are necessary. This requires binding design decisions (i.e., modularization) but also the need for consis-

tent software documents (e.g., code or models). Furthermore, it is very important to follow coding guidelines, use consistent code styles, or adhere to interface specifications.

- **Multi Languages support:** Globalisation affects business and the application that are operating the applications that need to support multiple languages.
- Understandability (Code), Readability: Attributes of software to be understood by the developer, tester, or maintainer. (Boehm). Similar to changeability and modularization the code or model of a system needs to be easily understandable by the developers and architects – especially in large, distributed, and multilingual teams.

5.2.2 Efficiency

The set of attributes that focus on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

- **Time Behaviour:** Attributes of software that bear on response and processing times and on throughput rates in performing its function. (ISO 9126: 1991, A.2.4.1) Business applications require a high level of time-efficiency since a company's main business processes may depend on the application. Slow applications that implement critical processes, will slow down the hole organisation.
- **Resource behaviour:** Attributes of software that bear on the amount of resources used and the duration of such use in performing its function. (ISO 9126: 1991, A.2.4.2). An important requirement for business applications is scaling therefore the resource consumption of the overall system as well as the business applications needs to be considered. In order to provide SME applications it is essential that software grows with the organisation.

5.2.3 Reliability

The set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.

- **Fault Tolerance:** Attributes of software that bear on its ability to maintain a specified level of performance in cases of software faults or of infringement of its specified interface. (ISO 9126: 1991, A.2.2.2). Business applications require a high level of stability since companies depend on the application. While failures are always possible, but a failure in one module should not stop the hole application.
- **Maturity:** Attributes of software that bear on the frequency of failure by faults in the software. (ISO 9126: 1991, A.2.2.1). Business applications require a high level of stability since companies depends on the application. Therefore the code should be mature.
- Recoverability: Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it. (ISO 9126: 1991, A.2.2.3).

5.2.4 Portability

The set of attributes that bear on the ability of software to be transferred from one environment to another.

- **Adaptability:** Attributes of software that bear on the opportunity for its adaptation to different specified environments without applying other actions or means than those provided for this purpose for the software considered. (ISO 9126: 1991, A.2.6.1).

Adaptability is a very important characteristic for standard software providers that are required to build in as much flexibility to adapt to specific customers and customer problems as required. However archiving that goal is very difficult since the demand for adoption by customers is difficult to foresee, especially in the more heterogeneous SME market. The need for Adaptability required SAP to ship the platform (SAP ERP…) including the source codes to allow customers to modify and adapt (more related to QA - Changeability). However for SMEs those kinds of adaptation will be too cost intensive. However VIDE models should allow leveraging changes/modification to configurations for applications with a higher level of adaptability.

- Installability: Attributes of software that bear on the effort needed to install the software in a specified environment. (ISO 9126: 1991, A.2.6.2).
- Replaceability: Attributes of software that bear on the opportunity and effort of using it in the place of specified other software in the environment of that software. (ISO 9126: 1991, A.2.6.4)

5.2.5 Functionality

The set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.

- **Suitability:** Attribute of software that bears on the presence and appropriateness of a set of functions for specified tasks. (ISO 9126: 1991, A.2.1.1). As outlined in section 5.3.1.1 security is an important issues for business applications.
- **Accuracy:** Attributes of software that bear on the provision of right or agreed results or effects. (ISO 9126: 1991, A.2.1.2). Accuracy of results is certainly one of the most important quality characteristic of business software and directly effected by behavioural models.
- Interoperability: Attributes of software that bear on its ability to interact with specified systems. (ISO 9126: 1991, A.2.1.3). While interoperability is important especially for collaborative business applications as provided by the SOA architecture. However interoperability depends to a large extend on interoperability of static structures such as data types. Interoperability is influenced less by the behavioural models therefore it is not considered especially important for the scope of the project.
- **Security:** Attributes of software that bear on its ability to prevent unauthorized access, whether accidental or deliberate, to programs and data. (ISO 9126: 1991, A.2.1.5). The information that is stored in enterprise systems is often the major asset of an enterprise that needs to be protected.

5.2.6 Usability

A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users. *The effort needed for use, and on the individual assessment of such use, by a stated or implied set of users*

- Understandability (System): Attributes of software that bear on the users' effort for recognizing the logical concept and its applicability. (ISO 9126: 1991, A.2.3.1).
- Learnability: Attributes of software that bear on the users' effort for learning its application (for example, operation control, input, output). (ISO 9126: 1991, A.2.3.2).

- Operability: Attributes of software that bear on the users' effort for operation and operation control. (ISO 9126: 1991, A.2.3.3).

The Quality Attribute Usability refers to the Usability of an application build using the VIDE languages. Since the project is focussing on the backend implementation and cared less about user interface issues. The Quality Attribute Usability can be neglected when evaluation the quality of VIDE code. Certainly the tools for creating VIDE code are obliged to usability.

5.3 Sources for Domain specific quality defects

The following programming languages and techniques are important in the SAP development environment: ABAP, ABAP-OO, Java, HTML, BSP, Microsoft Visual Basic and C/C++. HTML and BSP are both language used for generating user interfaces, Microsoft Visual Basic is used to integrate with Microsoft products, such as Microsoft Office, and the major use of C/C++ is for implementing basic elements, such as the APAP compiler. None of these languages are used for the implementation of business logic within SAP and therefore of less interest.

Business logic is implemented using ether Java or ABAP. Since Java is a general purpose language and APAB has been especially developed for implementing business application, including building mechanism for database manipulation and queries that are both essential part of the VIDE language, we'll focus on ABAP for the extracting quality defects. **ABAP** (**A**dvanced **B**usiness **A**pplication **P**rogramming) is a high level 4GL programming language invented and used by SAP to implement all kinds of business applications on top of databases.

Since the VIDE language defines model level programming semantics, one can expect quality defects to be similar – respective subclasses – of quality defects on the code level. Therefore this section derives the domain specific model defects from existing tools and guidelines (code guidelines, naming conventions, database access ...) for the ABAP language. Testing distinguishes between *static* and *dynamic* tests (Perry, 2000). A dynamic test the execution of a program with some test data is tested, while a static test used the static definitions of a program (usually the code, documentation…) for testing. Since the VIDE models itself – that means without generated code or model simulation - are static definitions we'll focus on static testing in the next sections.

5.3.1 (Development) Guidelines

Guidelines are a very well known and common mechanism to ensure unification and compliance for development artefacts such as code, security policies or documentation. Many of the existing defects/fault models originate from such guidelines that have been modified and formalized in order to automate them for fault detection. An important source for business specific defect models are therefore guidelines for creating business applications that will be investigated in this chapter.

5.3.1.1 Security Guidelines

Security certainly is important for business applications. SAP provides comprehensive documentation about how to develop secure ABAP (SAP, 2005a) and Java applications (SAP, 2005b). The guidelines address topics such as cross-site scripting (XSS), SQL injection, input validation, URL encoding, secure data storage, logging, virus scanning, and more. For each topic the security vulnerability is described and if any standard solutions from the SAP NetWeaver platform exist this is presented, including functions and interfaces that need to be used. If no solution is available from the SAP NetWeaver platform, recommendations are

given about appropriate security measures to take. This chapter will outline the security guidelines considered important for VIDE from the ABAP Security Guidelines (SAP, 2005a). A complete list of security guidelines can be found in (SAP, 2005a, 2005b).

Passwords are used for user authentication to protect applications. Dealing with passwords in favour of other authentication mechanisms, such as smartcards, requires some considerations. For instance passwords …

- should not be saved or transmitted in plaintext,
- should not be hard-coded in the source code,
- should not be recorded in log/protocol/trace files,
- …

Cross-site Scripting (XSS) and SQL injection are well known attack mechanisms often seen for collaborative websites where is user is allowed to edit the content. The edits are then processed by a program. If the processing is done without verification this mechanism may be used to insert malicious code, such as JavaScript or SQL, into the code based. A golden rule of thumb is therefore to *"never trust any information coming from the outside, and never assume anything about it"* (SAP, 2005a). Whenever software processes input from various sources, e.g.

- User input from a GUI ,
- Parameters from a configuration file,
- Data from a database,
- Data from remote function calls,

it should make sure that this input is in the expected form. This may be enforced by calling appropriate check method. Their use may be checked using defect detection.

5.3.2 Programming style

Most Programming languages allow for different programming styles that do not influence the semantics of the program. For instance usually indentation has no effect on the programming semantics (except for Python programs) but helps a programmer to understand the code better. Programming style-guides usually include instruction for the use of comments, naming conventions and the use of indentations. Code conventions are often specific to different programming languages. They usually cover naming conventions (filename, class name, variable names…), indentation, commenting, declarations, statements, white space and good programming practices. A list of naming conventions for specific languages may be found in (Wikipedia, 2007).

For VIDE many of the programming style guides only effect the textual syntax (e.g. indentations) However some programming style guides also effect the visual syntax (e.g. naming conventions) or even instances of the meta model (e.g. Naming conventions, proper commenting) and may therefore be enforced by defect/fault detection.

Some examples of programming style guides for the ABAP language are taken from (Blumenthal & Keller, 2006a, 2006b, 2006c; Heuvelmans et al., 2003).

5.3.2.1 Naming conventions

Naming convention defined rules for character sequences to be used for identifiers in source code and documentation. Naming convention increase aim to increase the consistency of source code and documentation for easier reading, understanding and improved source code

appearance. An example for naming conventions for ABAP code are conventions for *external names*. *External name* are repository objects with public visibility (Blumenthal & Keller, 2006c):

- CL_<name> for global classes
- IF_<name> for global interfaces
- CX_<name> for exception classes
- CL_OS_<name>, IF_ OS_<name> and CX_ OS_<name> should be used for Object Services
- CL_BADI_<name>, IF_ BADI _<name> and CX_ BADI _<name> should be used for Object Services

The naming conventions should be not be used for *internal names* to indicate what is visible form other Objects. Other naming conventions are related to languages issues. Usually the English language has to be used for code identifiers as well as for comments.

5.3.2.2 Source code sequence

Programming language guidelines often recommend a certain structure or sequence of programming artefacts. For instance the structure of Java programs is usually

1. Package declaration
2. Import
3. Class declaration
4. Class attributes
5. Class methods
    a. Constructor
    b. Main method

Also within method implementation programmers should follow given structures and coding sequences. Example for recommendation on those structures can be found for instance in (Blumenthal & Keller, 2006a) which recommends sequences for the sequence of

- **Declaration vs. Implementation** – Declarations (like imports, interface definitions) and implementation should be separated. Usually declarations are first and followed by the implementations.
- **Sequence of program parts** – Declarations or top down approach recommended. Bottom up means that within a program things are defined before they are used. Top down means that the program is structured among the importance, e.g. main components (interfaces, classes …) come first and are followed by helper classes. The selected approach should be used consistently within any given program.
- **Sequence of declarations** – The sequence of component declarations should also be done in a consistently. For example 1. Types 2. Constants 3. Static components 4. Instance components and 5. Field symbols
- **Sequence of statements in procedures (or methods)** – Procedures (or methods) should also be structured in a consistent manner. For ABAP programs (but also for other programming languages such as Java) it is recommended start an implementation with local declarations (types, local variables…) followed by functional statements.

5.3.2.3 Avoid outdated programming constructs

As programming languages and libraries evolve some programming constructs or interfaces are replaced with other (better) programming construct or interfaces. In order to stay compatible with legacy code the *old* programming constructs or interfaces valid for a time. In Java for instance library method are marked *deprecated* which mean that they may be removed/replaced in further releases. Java complier options allow checks for deprecated methods which results in a warning. Other programming languages do not have such a build-in support. Therefore ether guidelines on how to deal with legacy and new code are give or methods for automatic defect detection are used. An example for the ABAP languages is the use of binary operators (Blumenthal & Keller, 2006a). ABAP supports relational operators (=, < >, <, >, <=, =>) as well as character operators (EQ, NE, LT, GT, GE) to express binary operators. Relational operators are more readable therefore their use is recommended.

5.3.2.4 Guidelines specific for ABAP-OO

The ABAP language evolved from a functional programming language in to an Object Oriented programming language ABAP-OO. With the appearance of the OO concepts and some language extensions a couple of recommendations are given in (Blumenthal & Keller, 2006b). For instance Blumenthal and Keller recommend restrictive interface design that should be easy only when needed. They recommend to

- Declare classes as final
- Restriction of the number of public components. Components that can be private should be declared private
- Attributes that are declared public should be READ-ONLY
- Consider private instantiation of class (CREATE_PRIVATE and offer factory methods)

5.3.3 Tool based defect detection

The ABAP development environment contains a couple of analysis tools (Eilenberger & Schmitt, 2003):

- The **ABAP Debugger** is a debugger for the ABAP language used for dynamic analysis such as bug detection.
- The **Runtime Analysis** tool allows analysis of the duration and performance of ABAP code, from individual statements up to complete transactions.
- The **Coverage Analyzer** is a monitor for tracking how often a processing block was executed. The tools is for dynamic analysis of running systems and used to exhibit unreached code blocks
- The **Runtime Monitor** is an instrument that supports the recording of information on user triggered events that can be used to replay user session.
- The **Memory Inspector** is used to analyse memory snapshots, e.g. the result of a core dump
- The **ABAP Unit** are unit tests (Perry, 2000) for the ABAP language. It is used to define and verify test cases for unit test, but doesn't contain any kind of generic defects of smells, but specific test cases.
- The ABAP **Code Inspector** is used for static analysis of ABAP code.

Since defect analysis and fault detection focuses on the static analysis we'll focus on tools for static analysis of ABAP coding which is in this case the ABAP code inspector.

5.3.3.1 ABAP Code Inspector

The *ABAP Code Inspector* is a tool for static code analysis of ABAP code intended to *"help you easily identifying some of these types (main focus on syntax, performance and security) of shortfalls" (Eilenberger & Schmitt, 2003)*. Figure 7 below shows a screenshot of the tool showing the results of an analysis.

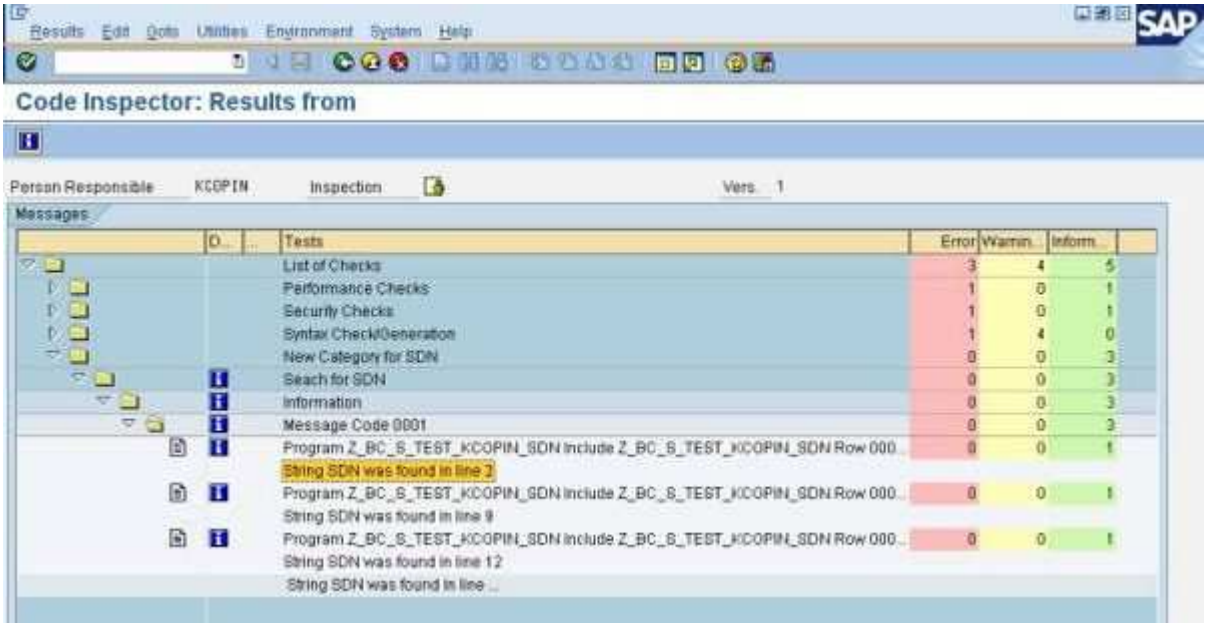

Figure 7. *Result screen ABAP Code Inspector*

The tool allows checking all kinds of ABAP code, such as programs, function groups or classes. The system can be extended to support additional check. However more interesting are the standard checks that come with the system. The build-in checks fall into three categories

- Syntax checks
- Security checks and
- Performance checks

that are described in more detail below. The tools also allows to defines so called *Search Options* that allows the definition of search patterns to test compliance to for instance the naming rules described above.

5.3.3.2 Syntax check

The first level of syntax checks are "normal" syntax checks for the ABAP languages that are similar to syntax check other language parsers, such as the parser developed for the textual VIDE syntax. Those checks are of little interests for defect/fault detection since they are already checks by the language parser.

- **References to program external units:** Verifies if external program units (e.g. subroutine calls) exist and interfaces are used correctly.
- **Multi-language enabling:** Searches for constructs that hamper the use of a program in different languages — for example, text literals without text IDs. Text literals appear in the language in which they were typed, and are not processed by translation services.
- **Package check:** Detect the illegal use of objects from other packages. This check is often performed by a compiler.

- **Character portability (EBCDIC/ASCII portability):** Detects whether a program behaves differently in EBCDIC (Extended Binary Coded Decimal Interchange Code) and ASCII (e.g., the comparison of character fields). This maybe extended to other character encodings, e.g. Unicode.
- **Generation limits:** Determines if generation limits, such as the maximum number of objects, are close to being reached. This check depends on transformation rules and limit of the targeted platform.
- **Statements in wrong context:** Scans for statements that are used in an inappropriate language context. For example, the COMMIT WORK statement within a SELECT ... ENDSELECT loop leads to the loss of the database cursor.
- **Unnecessary items:** Searches for form subroutines that are not used in a program, or fields that do not have read access. This check is often performed by a compiler.

5.3.3.3 Security checks

Some ABAP statements can endanger the stability, data integrity and security of the overall system. The Code Inspector therefore performs a couple of security checks to detect critical coding. The checks performed are listed below.

- **Internal statements:** ABAP supports so called internal statements intended only for internal use by SAP. Their signature may change without notice and should therefore not be used in programs.
- **Authority checks:** For better performance SAP systems enforce automatic authority checks only for programs (SAP transactions) called directly by a user. Automatic authority checks for function calls within programs need to be implemented by the programmer, which is checked by this check. In VIDE those checks may also be enforced using AOP (see (Vide, 2007b)).
- **Database operation:** Some ABAP statements potentially risk the portability of the code to other database systems (native SQL statements via EXEC and database hints) or the data integrity (ROLLBACK WORK). Therefore security checks through warnings for those statements. Introducing native SQL into ABAP code is specific to ABAP. However the opaque expression in Action Semantics similar introductions of native code that should be checked to increase the portability of VIDE code.
- **Repository objects:** SAP systems store all development fragments (e.g. programs, screens, global types…) as repository objects in the database. From which they may be retrieved (e.g. READ REPORT) afterwards. However this should only be done by internal development tools. This behaviour is specific for SAP systems and therefore unimportant for VIDE coding.
- **Access to database tables:** Database tables may contain confidential information, such as personal data. Therefore their access should be restricted and performed only when the access is authorized. The check allows the specification of critical database tables and check if access to the data is only done after authorisation. In VIDE those checks may also be enforced using AOP (see (Vide, 2007b)).
- **Handling system return codes:** Not handling return codes (e.g. if the method failed) may be suspicious and is therefore checked.

5.3.3.4 Performance checks

Performance checks are usually done using dynamic testing methods. However some static programming constructs are know to be performance critical. The Code Inspector implements a couple of those checks focussing on inefficient database queries (SQL) such as for example WHERE clauses that do not use an existing database index. Those checks are specific to SQL as query language. Therefore they can not be easily adapted for the OCL queries using in the VIDE language. However badly designed OCL queries may also have a bad impact on the system performance and should therefore be verified. For instance (OCL) select maybe used similar to the (SQL) WHERE statements and may cause similar impacts on the system performance.

6 Resulting defect model for the business domain

After listing the main collections of quality defects and describing the domain specific requirements this section is concerned with the selection of domain-specific quality defects targeted in the VIDE project. As we have seen the most important characteristics of quality defects is their focus on non-functional problems in software models, and that they are project-independent, language-independent, symptom-based, and treatment-oriented.

The following list is compiled from the quality defects outlined in sections 4 combined with the domain and the quality characteristics for the domain from section 5.2 and 5.3. The following Table 54 is a selection of the most important and frequent quality defects in the data-oriented business domain. This selection was based on the following objective, and subjective criteria:

- The sum of interestingness (●-Dots) should include at least 2 full dots in sum (objective)
- The propability of the quality defect in a PIM should be high (subjective)
- It should be possible to associate concrete treatments (e.g., refactorings) with the quality defect.
- The quality defect should not focus on problems that would make the model not compilable (e.g., duplicated attribute names in a class) and not conforming to a standard.

The selection serves as a basis and priority list for defect detection methods conceptualized during WP4 and reported in deliverable D4.2. The implementation of these detection techniques in the VIDE development environment will be conducted during WP9.

Table 54. *Selected Quality Defects targeted for VIDE WP9*

Name	Type of Quality Defect	Description	PIM level	Domain	Behavior	Local	Selection Rationale
Long method	Structure	A very large method.	●	○	●	●	Relation to the VIDE behavior model
Long parameter list	Structure	A method with too many parameters.	●	○	◎	●	Intersection between the VIDE behavior & structural models
Feature envy	Structure	A method is occupied more with data and methods in other classes than its own.	●	○	◎	◎	Intersection between the VIDE behavior & structural models
Duplicated Code	Control	Duplicated OCL expressions.	●	○	●	◎	Problems in OCL code
Message chains	Structure	One object asks another object for data in a third object (and so on).	●	○	◎	◎	Intersection between the VIDE behavior & structural models
Lazy class	Structure	A class that isn't doing much.	●	○	◎	●	Relation either to the behavior or structural model
The Blob (God Class)	Structural	Classes with too many functionality and associations to other classes.	●	○	◎	●	Intersection between the VIDE behavior & structural models
Data class	Structure	Classes that do almost exclusively store information for other classes. Optionally, these classes have getter and setter methods for the attributes.	●	●	○	●	Data orientation & Problems in structural models
Data clumps	Structure	Data items (i.e., attributes,	●	●	◎	○	Data orientation & Problems in

Name	Type of Quality Defect	Description	PIM level	Domain	Behavior	Local	Selection Rationale
		parameters, local variables, etc.) that appear in groups all over the system (e.g., id, surname, forename, salary).					structural models
Type Embedded in Name	Semantic	Type information is redundantly encoded in the name / identifier of an attribute, method, etc.	●	⊙	⊙	●	Problems in identifier
Uncommunicative Name	Semantic	The name does not communicate the intent (e.g., short names, abbreviations, …).	●	⊙	⊙	●	Problems in identifier
Inconsistent Names	Semantic	Names are not consistent throughout the system.	●	⊙	⊙	○	Problems in identifier
Complicated Boolean Expression	Control	Complex condition involving Boolean operators (“and”, “or”, “not”).	●	●	●	●	Problem in Code & Relation to the VIDE behavior model (optionally, in OCL code)
Combinatorial Explosion	Control	Code duplication for many slightly different features (e.g., queries)	●	⊙	●	○	Problem in Code & Relation to the VIDE behavior model (optionally, in OCL code)
Conditional Complexity	Control	Large and complex conditional statements (i.e., if, switch etc.)	●	⊙	●	●	Problem in Code & Relation to the VIDE behavior model (optionally, in OCL code)
Magic Literal	Control	Numeric or string literal that appears in the middle of an OCL expression without explanation.	●	⊙	●	●	Problem in Code & Relation to the VIDE behavior model (optionally, in OCL code)
Redundant data	Data	Data is stored in different places (e.g., birthday).	⊙	●	⊙	⊙	Data orientation & Problems in identifier
Tables with too many columns	Data	Items consist of too much data and table probably lacks cohesion.	⊙	●	○	●	Problems in data bases (resp. Persistence layer)
Coupling	Structure, Control	Parts are linked by an extensive network of data or control flows.	●	⊙	⊙	●	Intersection between the VIDE behavior & structural models
9. Minimize the number of bubble types	Structure	A diagram that holds more than six elements (bubbles).	●	○	○	●	Problems in Diagrams
10. Include White Space in a diagram	Layout	Elements in a diagram that are too close together	●	○	○	●	Problems in Diagrams
16. Reorganize large Diagrams into several smaller ones	Structure	Diagram is too large	●	○	○	●	Problems in Diagrams
26. Apply color or different fonts sparingly	Layout	More than six colors in a single diagram	●	○	○	●	Problems in Diagrams
27. Describe diagrams with notes	Structure	Missing comments / notes about the diagram	●	○	○	●	Problems in Diagrams
158. Strive for left to right ordering of messages	Layout	Message flow that is unordered and makes a zig-zag.	●	○	○	●	Problems in Diagrams

These 25 quality defects represent the currently most interesting ones that will be targeted in the realization of the quality defect diagnosis during WP9. If all of the quality defects will be used is still uncertain – for example, quality defects that affect multiple locations (e.g., Inconsistent Names) or unused elements (e.g., “Strive for left to right ordering of messages” when no sequence diagrams are used) might be excluded.

7 Concluding Remarks

In this report we presented an extensive overview of existing quality defects affecting quality aspects of software products, processes, projects, and organizations as well as techniques for their diagnosis (presented in D4.2) based on a systematic literature review. This review was used to summarize the existing literature and construct an objective and comprehensive overview about quality defects, related concepts, and their diagnosis techniques. We selected more than 560 black and grey publications published in scholarly literature, identified 43 concepts with quality defects, and listed 800 quality defects in this report. We classified, evaluated, and discussed the research on the quality defects based on the quality defect description, their types and different criteria such as the type of software artifact concerned, the process they are embedded into, and the quality aspect affected. The results of the work in WP 4 of the VIDE project is as follows:

- In contrast to the insular and inconsistent collections in other publications this report presents the results of a systematic literature review to create a comprehensive and uniform collection of these quality defects and to start a quality defect body of knowledge.
- The collection of definitions of existing quality defect related concepts and the synthetization of a consistent and uniform definition of quality defects.
- The analysis of existing quality defects regarding their applicability in the context of Model-driven Software development.
- The construction of an information model based on UML 2.0 that describes the information that might be used to diagnose quality defect in MDSD models and especially PIMs.
- The selection of quality defects that should be diagnosed in the VIDE environment in order to support the modelers during their design activities. This information will be used in WP9 to design and realize the diagnosis techniques.
- Finally, the identification of gaps in the current research and body of knowledge in order to support where future research is needed.

7.1 Recommendations

Furthermore, we identified important open research issues that remain to be solved. In summary, we identified the need for an comprehensive ontology to systematize the defect corpus, a naming taxonomy to equalize and systematize the names, an formalization of the quality defects based on different languages and environments (e.g., MDSD, OO, AOP, etc.), as well as specific defect diagnosis techniques for their discovery. Additionally, more empirical evidence s required about the precise effects of the quality defects on the quality aspects on the models and the resulting software systems.

7.2 Outlook

Researches in software engineering are more and more equipped with techniques and method for the systematic identification of symptoms, diagnosis and prognosis of quality defects, and indication of treatments and preventive measures. Nevertheless, in order to handle the increasing amount of knowledge about software systems more techniques of the diagnosis of quality defects on all levels of software products, processes, projects, and organizations are required. As the field of software engineering matures and the possibilities for more advanced diagnosis

and prognosis techniques increase, the field of software quality assurance based on quality defects promises to be an exciting area for future research.

The sister-report D4.2 will include a summary of quality defect diagnosis techniques, their characteristics, benefits, and shortcomings. Beside the diagnosis techniques it will include visualization concepts for quality defects in MDSD and the information model based on UML 2.0 and compares it with available information from other environments such as eclipse-UML, or Java.

The work package WP9 will be used to realize the diagnosis and visualization techniques for QDs in the VIDE environment.

8 Glossary

Analyst / Designer: Analysts/Designers are responsible for the conceptual model of business entities and the high level business logic. They use design artefacts and models produced by the business analyst and transforms them into a design. Analysts/Designers work on PIM level in the VIDE tool stack.

Analyst/VIDE Programmer: The Analyst/VIDE Programmer is responsible for the completion of the behavioural model to allow model simulation (i.e. for testing) and the transformation of the models into code. Analysts/VIDE Programmers work on PIM level in the VIDE tool stack.

AOP: **A**spect-**O**riented **P**rogramming is a programming paradigm that attempts to aid programmers in the separation of concerns, specifically cross-cutting concerns, to advance the modularization of software. AOP uses crosscutting expressions that encapsulate the concern in one place.

Architect: The architect is responsible for building the transformations of the behavioural models described using VIDE into platform specific coding. The architect is an expert in the target platform (i.e. Struts, …) and the programming language (i.e. Java) but also has a sufficient understanding of UML and VIDE to be able to define the transformation. Architects work on PIM&PSM level in the VIDE tool stack.

ATL: The **A**TLAS **T**ransformation **L**anguage is a result of the MODELWARE project. This transformation language is closely related to the QVT standard and provides a running implementation.

BPMN: Business Process Modelling Notation. The OMG standard BPMN provides a notation that is understandable by business users, including business analysts (creating the initial drafts of the processes), the technical developers (responsible for implementing the technology that will perform those processes), and the business people (who will manage and monitor those processes).

Business Analyst: The Business Analysts advise enterprises on analysis, conception and implementation of IT solutions. They constitute the connection between the customer and the involved IT specialists and need technical as well as social competences. Business Analysts work on CIM level in the VIDE tool stack.

CIM: A **C**omputation **I**ndependent **M**odel represents the user requirements in an abstract, high level view on a software or business system. The transition of a CIM Model into a Platform Independent Model (PIM) should be done automatically using a model transformation.

Domain User (Customer): The Domain User is the end user of the constructed software solution. He works for the customer and is an expert in his special domain typically without knowledge technical issues. The Domain User works on CIM level in the VIDE tool stack.

EMF: **E**clipse **M**odeling **F**ramework is a modelling framework for building tools and other applications based on a structured data model. EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. EMF provides the foundation for interoperability with other EMF-based tools and applications.

GEF: **G**raphical **E**diting **F**ramework allows developers to create a rich graphical editor from an existing application model. Developer can take advantage of many common opera-

tions provided in GEF and/or extend them for the specific domain. GEF employs an MVC (model-view-controller) architecture which enables simple changes to be applied to the model from the view.

GMF: The **G**raphical **M**odeling **F**ramework provides a generative component and runtime infrastructure for developing graphical editors based on the Eclipse Modeling Framework (EMF) and Graphical Editing Framework (GEF).

IDE: Integrated **D**evelopment **E**nvironment assists computer programmers in developing software usually consisting of a source code editor, a compiler and/or interpreter, build-automation tools, and a debugger. The VIDE project will extend an existing IDE with tools for describing UML2 Action Semantics

M3/M2/M1 Layers: Metamodelling is defined into a four-layered architecture. The M3 layer provides a meta-meta-model at the top layer. This M3-model is the language used by MOF to build meta-models, called M2-models. These M2-models describe elements of the M1-layer, and thus M1-models. The M0-layer is used to describe the real-world.

MDA: Model-**D**riven **A**rchitecture is a software design approach intended to support model-driven engineering of software systems. MDA was initiated by the OMG.

MDST: Model **D**riven **S**oftware **T**esting derives test cases in whole or in part from a model that describes some (usually functional) aspects of the test system. In VIDE testing should be supported on model (e.g. model simulation) and code level verify the correctness of code transformations.

ModelBus: ModelBus are tools dedicated to model driven development developed by the MODELWARE project. The key feature of ModelBus is possibility to exchange models in heterogeneous formats and a transparent integration of model based tool.

MOF: Meta-**O**bject **F**acility is standard for Model Driven Engineering, proposed by the OMG. MOF provides a meta-meta-model at the top layer and means to create and manipulate models and meta-models. There are two relevant versions of this standard, MOF 1.4 (Object Management Group 2002) and MOF 2.0 (Object Management Group 2004).

OCL: Object **C**onstraint **L**anguage. OCL statements serve as the most precise means of model specification within the UML and MOF model and meta-model definitions. For that purpose OCL was defined to be able to express constraints for any kind of UML elements. OCL moreover provides means to express any (first-order) query on some instance of a UML class diagram.

OMG Object **M**anagement **G**roup (OMG) is a consortium, originally aimed at setting standards for distributed object-oriented systems, and is now focused on modelling (programs, systems and business processes) and model-based standards in some 20 vertical markets.

Petri Net: Petri Nets are a formal, graphical, executable technique for the specification and analysis of concurrent, discrete-event dynamic systems; a technique undergoing standardization, initially developed by C. A. Petri for the specification of concurrent (parallel) systems.

PIM: A **P**latform **I**ndependent **M**odel is a model of a software or business system that is independent of the specific technological platform used (PSM Level) to implement it. The transition of a PIM Model into a Platform-specific (PSM) model should be done automatically using a model transformation.

PSM: A **P**latform **S**pecific **M**odel is a model of a software or business system that is linked to a specific technological platform (e.g. a specific programming language, operating

system or database). The PSM Model should allow for an automatic transformation into code.

Query: A query is the extraction of data from a structured source of information. In the VIDE context, queries are sub-expressions of the VIDE language which extract data from a UML class diagram.

QVT: Query / **V**iews / **T**ransformations is an emerging OMG standard provides technology neutral solutions for querying, transforming and specifying views of MOF-based models.

SDL: The **S**pecification and **D**escription **L**anguage is a specification language for describing system behaviour. Its major use case is in the telecommunication industry for descriptions of process control and real-time applications.

SME: Small & **M**edium-sized **E**nterprises is an abbreviation to classify companies whose headcount or turnover falls below certain limits.

Tefkat: Open source model transformation language developed at Queensland University.

User: A person who interacts with a system.

User Interface (UI): All aspects of a system with which a user can interact and perceive.

UML: Unified **M**odeling **L**anguage is a specification language for object modelling defined at the OMG. UML2 Action Semantics is an essential part of UML 2.0 for the VIDE project.

UML Action semantics: UML Action Semantics refers to the capabilities of UML to describe behaviour algorithmically. UML Action Semantics were in UML 1.4 separated from the rest of UML; since UML 2, one should rather speak of the behavioural part of UML (which is sub-divided in UML actions, activities, and behaviour). Contrary to its name, UML Action Semantics, does primarily define an abstract syntax rather than semantics.

Visual Design: The portion of a user interface that is concerned with the aesthetic quality of an application. Composed of variables that address a specific purpose or function, such as font, color, and images, which impact the appearance, organization and layout of the graphical elements in a user interface.

XMI: XML **M**etadata **I**nterchange is a MOF-based specification providing the rules of XML serialization of models, allowing their transfer between standard-compliant tools.

9 References

Abreu, F. B. E. (1997). Pedagogical patterns: picking up the design patterns approach. *Object Expert, UK * vol 2 (March April 1997), no 3, p 37, 41, 3 refs.*

Alexander, R. T., Offutt, J., & Bieman, J. M. (2002). *Syntactic fault patterns in OO programs.* Paper presented at the Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pages 193-202.

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

Amberg, M., & Schumacher, J. (2002). CRM-Systeme und Basistechnologien. In M. Meyer (Ed.), *CRM-Systeme mit EAI - Konzeption, Implementierung und Evaluation* (pp. 21--59). Wiesbaden: Vieweg.

Ambler, S. W. (2006). *The Elements of UML 2.0 Style* (1st Edition): Cambridge University Press.

Ambler, S. W., & Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design* (1st Edition (March 3, 2006)): Addison-Wesley Professional.

Andrea, J., Meszaros, G., & Smith, S. (2002). *Catalog of XP Project 'Smells'*. Paper presented at the 3rd International Conference on XP and Agile Processes in Software Engineering (XP 2002), Alghero, Sardinia, Italy, pages 130-133.

ArgoUML. (2007). ArgoUML User Manual - chapter 15. Retrieved 15. June, 2007, from http://argouml-stats.tigris.org/documentation/manual-0.24/ch15.html

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.

Baldwin, K., Gray, A., & Misfeldt, T. (2006). *The Elements of C# Style* (1st Edition): Cambridge University Press.

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

Baumeister, J., Puppe, F., & Seipel, D. (2004). *Refactoring Methods for Knowledge Bases.* Paper presented at the 14th International Conference on Engineering Knowledge in the Age of the Semantic Web (EKAW), pages 157–171.

Becker, P. (2000a). Common design mistakes, part 1. *The C/C++ Users Journal, 18*(1), 73-78.

Becker, P. (2000b). Common design mistakes, part 2. *The C/C++ Users Journal, 18*(2), 77-84.

Bennett, K. H., & Rajlich, V. T. (2000). *Software Maintenance and Evolution: A Roadmap.* Paper presented at the Future of Software Engineering Track of 22nd ICSE, Limerick, Ireland, pages 73-87.

Biolchini, J., Mian, P. G., Natali, A. C. C., & Travassos, G. H. (2005). *Systematic Review in Software Engineering* (No. RT-ES 679/05). Rio de Janeiro: Systems Engineering and Computer Science Department, COPPE/UFRJ.

Bloch, J., & Gafter, N. (2005). *Java Puzzlers: Traps, Pitfalls, and Corner Cases* (1st Edition): Addison-Wesley Professional.

Blumenthal, A., & Keller, H. (2006a). An insiders guide to writing robust, understandable, maintainable, state-of-the-art ABAP programs - Part 1. *SAP Professional Journal, Wellesley Information Services, Jan./Feb*, 3--26.

Blumenthal, A., & Keller, H. (2006b). An insiders guide to writing robust, understandable, maintainable, state-of-the-art ABAP programs - Part 2. *SAP Professional Journal, Wellesley Information Services, March/April*, 3--26.

Blumenthal, A., & Keller, H. (2006c). An insiders guide to writing robust, understandable, maintainable, state-of-the-art ABAP programs - Part 3. *SAP Professional Journal, Wellesley Information Services, May/June*, 3--28.

Booch, G. (2007). Website of the Handbook of Software Architecture - Pattern Section. Retrieved 11. July, 2007, from http://www.booch.com/architecture/patterns.jsp

Boundy, D. (1993). Software cancer: the seven early warning signs. *Software Engineering Notes (SEN), 18*(2), 19.

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.

Brown, W. J., McCormick, H. W., & Thomas, S. H. (1999). *AntiPatterns and Patterns in Software Configuration Management* (1st Edition): John Wiley & Sons, Inc.

Bruntink, M., van, D. A., Tourwe, T., & van, E. R. (2004). An evaluation of clone detection techniques for crosscutting concerns. *Proceedings. 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11 14 Sept. 2004 * Los Alamitos, CA, USA: IEEE Comput. Soc, 2004, p 200 9*.

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

Buck-Emden, R., & Zencke, P. (2004). *mySAP CRM: The Offcial Guidebook to SAP CRM Release 4.0*: SAP Press.

Buschmann, F., Henney, K., & Schmidt, D. C. (2007). *Pattern-oriented Software Architecture: On Patterns and Pattern Languages* (Vol. 5). New York: John Wiley & Sons, Inc.

Cheung, S.-C., & Kramer, J. (1993). *Tractable Flow Analysis for Anomaly Detection in Distributed Programs.* Paper presented at the 4th European Software Engineering Conference on Software Engineering (ESEC/FSE), Garmisch-Partenkirchen, Germany, September 13 - 17, 1993, pages 283-300.

Choi, S.-E., & Lewis, E. C. (2000). *A study of common pitfalls in simple multi-threaded programs.* Paper presented at the Thirty-first SIGCSE technical symposium on Computer science education (SIGCSE), Austin, Texas, United States, pages 325-329.

Ciolkowski, M., Laitenberger, O., Rombach, D., Shull, F., & Perry, D. (2002). *Software inspections, reviews and walkthroughs.* Paper presented at the 24th International Conference on Software Engineering (ICSE 2002), New York, NY, USA, Soc., pages 641-642.

Coad, P., & Edward, Y. (1993). *Object-oriented Design* (1st Edition): Prentice Hall.

Coad, P., & Nicola, J. (1993). *Object-oriented Programming* (1st Edition): Prentice Hall.

Cockton, G., & Gram, C. (1996). *Design Principles for Interactive Software* (1st edition (June 30, 1996)): Springer.

Coelho, W., & Murphy, G. (2007). ClassCompass: A software design mentoring system. *7*(1), 2.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development* (1st Edition (March 1, 2004)): Addison-Wesley Professional.

Copty, S., & Shmuel, U. (2005). *Multi-threaded testing with AOP is easy, and it finds bugs!* Paper presented at the 11th European Conference on Parallel Processing (EUROPAR), Lisbon Portugal, 30 Aug.-2 Sept. 2005, pages 740-749.

Correa, A. L., & Werner, C. (2004). *Applying Refactoring Techniques to UML/OCL Models.* Paper presented at the 7th International Conference on the Unified Modeling Language: Modeling Languages and Applications (UML), Lisbon, Portugal, October 11-15, 2004, pages 173-187.

Daconta, M. C., Monk, E., Keller, J. P., & Bohnenberger, K. (2000). *Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs* (1st Edition): John Wiley & Sons.

Daconta, M. C., Smith, K. T., Avondolio, D., & Richardson, W. C. (2003). *More Java Pitfalls: 50 New Time-Saving Solutions and Workarounds* (1st Edition). Indianapolis Indiana: Wiley Publishing Inc.

Demeyer, S., Ducasse, S., & Nierstrasz, O. M. (2003). *Object-oriented reengineering patterns*. San Francisco: Morgan Kaufman Publishers.

Deursen, A. v., Moonen, L., Bergh, A. v. d., & Kok, G. (2001). *Refactoring Test Code.* Paper presented at the Second International Conference on Extreme Programming and Flexible Processes (XP), pages 92-95.

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

Dudney, B., Krozak, J., Wittkopf, K., Asbury, S., & Osborne, D. (2002). *J2EE Antipatterns* (1st Edition): John Wiley & Sons, Inc.

Dudney, B., & Lehr, J. (2003). *Jakarta Pitfalls: Time-Saving Solutions for Struts, Ant, JUnit, and Cactus* (1st Edition (July 25, 2003)): John Wiley & Sons.

Eilenberger, R., & Schmitt, A. S. (2003). Evaluating the Quality of Your ABAP Programs and Other Repository Objects with the Code Inspector. *SAP Professional Journal, Wellesley Information Services, May/June*, 3--30.

Elssamadisy, A., & Schalliol, G. (2002). *Recognizing and responding to "bad smells" in extreme programming.* Paper presented at the 24th International Conference on Software Engineering (ICSE), Orlando FL USA, 19-25 May 2002, pages 617-622.

Farchi, E., Nir, Y., & Ur, S. (2003). *Concurrent bug patterns and how to test them.* Paper presented at the International Parallel and Distributed Processing Symposium (IPDPS), Nice France, 22-26 April 2003, pages 7 pp.

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

Fenton, N. E., & Ohlsson, N. (2000). Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering, 26*(8), 797-814.

Fincher, S., & Utting, I. (2002). Pedagogical patterns: their place in the genre. *SIGCSE Bulletin, USA * vol 34 (Sept. 2002), no 3, p 199 202, 18 refs.*

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

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

Galvans, A. (2006, 15 March 2006). Performance bug patterns and bug-hunting. Retrieved 1. June, 2007, from http://www.testingreflections.com/node/view/3398

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

Gibbon, C. A. (1997). *Heuristics for object-oriented design.* PhD Thesis, University of Nottingham, from http://www.cs.nott.ac.uk/~cah/pdf/cag-phd.pdf.

Glass, R. L. (2003). *Facts and Fallacies of Software Engineering*. Boston: Addison-Wesley Professional.

Green, R. (1996). How to Write Unmaintainable Code. Retrieved 21.11.2005, 2005, from http://mindprod.com/jgloss/unmain.html

Grotehen, T. (2001). *Objectbase Design: A Heuristic Approach.* PhD Thesis, University of Zurich, Zurich, from http://www.ifi.unizh.ch/ifiadmin/staff/rofrei/Dissertationen/Jahr_2001/thesis_grotehen.pdf.

Hallal, H. H., Alikacem, E., Tunney, W. P., Boroday, S., & Petrenko, A. (2004). *Antipattern-based detection of deficiencies in Java multithreaded software.* Paper presented at the Fourth International Conference on Quality Software (QSIC), Braunschweig Germany, 8-9 Sept. 2004, pages 258-267.

Hawkins, B. (2003). *Preventative Programming Techniques: Avoid and Correct Common Mistakes* (1st Edition): Charles River Media.

Heskett, J., Jones, T., Loveman, G., Sasser (Jr.), W., & Schlesinger, L. (1994). Putting the service-profit chain to work. In *Harvard Business Review* (pp. 164--174): Harvard Business Review.

Heuvelmans, W., A, K., B.Meijs, & Sommen, R. (2003). *Enhancing the Quality of ABAP Development*: SAP PRESS.

Hippner, H., Hoffmann, O., Rimmelspacher, U., & Wilde, K. D. (2006). IT Unterstützung durch CRM-Systeme am Beispiel von mySAP CRM. In H. Hippner & K. D. Wilde (Eds.), *Grundlagen des CRM, Second Editon* (pp. 15--44). Wiesbaden: Gabler.

Hippner, H., Rentzmann, R., & Wilde, K. D. (2004). Aufbau und Funktionalitaten von CRM-Systemen. In H. Hippner & K. D. Wilde (Eds.), *IT-Systeme im CRM: Aufbau und Potenziale* (pp. 13--42). Wiesbaden: Gabler.

Hippner, H., & Wilde, K. D. (2002). CRM - Ein Überblick. In S. Helmke, M. Uebel & W. Dangelmaier (Eds.), *Effektives Customer Relationship Management: Instrumente, Einführungskonzepte, Organisation* (pp. 3--37). Wiesbaden: Gabler.

Ho, W. J., Seung, G. K., & Chang, S. C. (2004). Measuring software product quality: a survey of ISO/IEC 9126. *IEEE Software, USA * vol 21 (Sept. Oct. 2004), no 5, p 88 92, 11 refs.*

Hovemeyer, D., & Pugh, W. (2004). *Finding bugs is easy*. Paper presented at the 19th annual conference on Object-oriented programming systems, languages, and applications (OOPSLA), Vancouver, BC, CANADA, pages 132-136.

Hovemeyer, D. H. (2005). *Simple and effective static analysis to find bugs.* PhD Thesis, University of Maryland at College Park, from https://drum.umd.edu/dspace/bitstream/1903/2901/1/umi-umd-2689.pdf.

Howard, M., LeBlanc, D., & Viega, J. (2005). *19 Deadly Sins of Software Security* (1st edition (July 26, 2005)): McGraw-Hill Osborne Media.

HPL. (2005). Hillside Pattern Library. Retrieved 10. Oct., 2005, from http://hillside.net/patterns/

IEEE-610. (1990). *IEEE Std 610.12-1990. IEEE standard glossary of software engineering terminology*: Institute of Electrical and Electronics Engineers.

IEEE-1044. (1995). *IEEE guide to classification for software anomalies* (No. IEEE Std 1044.1). New York, NY, USA: IEEE.

ISO. (2005). *ISO 9000: Quality management systems -- Fundamentals and vocabulary* (No. %(). #pub-ISO:adr#: ISO.

ISO/IEC-9126-1. (2003). *Software engineering: product quality. Part 1, Quality model* (Ed. 1.). Pretoria: International Organization for Standardization / International Electrotechnical Commission.

ISO/IEC-9126-3. (2004). *Software engineering: product quality. Part 3, Internal metrics* (Ed. 1.). Pretoria: International Organization for Standardization/International Electrotechnical Commission.

ISO/IEC-25000. (2005). *Software Engineering -- Software product Quality Requirements and Evaluation (SQuaRE) -- Guide to SQuaRE* (Standard).

ISO/IEC. (2000a). *TR 9126-2: Software engineering - Product quality - Part 2: External metrics*: ISO/IEC.

ISO/IEC. (2000b). *TR 9126: Software engineering - Product quality*: ISO/IEC.

Johnson, R. E., & Foote, B. (1988). Designing Reusable Classes. *Journal of OO Programming, 1*(2), 22-35.

Kasyanov, V. N. (2001). *A support tool for annotated program manipulation.* Paper presented at the Fifth European Conference on Software Maintenance and Reengineering (CSMR), pages 85-94.

Kataoka, Y., Ernst, M. D., Griswold, W. G., & Notkin, D. (2001). *Automated support for program refactoring using invariants.* Paper presented at the International Conference on Software Maintenance (ICSM), pages 736-743.

Kerievsky, J. (2005). *Refactoring to patterns* (1st Edition). Boston: Addison-Wesley.

Khan, K. S., Riet, G. t., Glanville, J., Sowden, A. J., & Kleijnen, J. (2001). *Undertaking Systematic Reviews of Research on Effectiveness: CRD's Guidance for those Carrying Out or Commissioning Reviews* (No. CRD Report 4, ISBN 1900640201): NHS Centre for Reviews and Dissemination, University of York.

Kitchenham, B. (2004). *Procedures for undertaking systematic reviews* (Technical Report No. TR/SE-0401). Keele: Department of Computer Science, Keele University and National ICT, Australia Ltd.

Koenig, A. (1989). *C Traps and Pitfalls* (1st Edition): Addison-Wesley Professional.

Kuranuki, Y., & Hiranabe, K. (2004). *AntiPractices: AntiPatterns for XP practices.* Paper presented at the Agile Development Conference (ADC), Salt Lake City UT USA, 22-26 June 2004, pages 83-86.

Laffra, C. (1996). *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips* (1st Edition): Prentice Hall.

Laitenberger, O. (2002). A Survey of Software Inspection Technologies. In *Handbook on Software Engineering and Knowledge Engineering* (Vol. II, pp. 517-555): World Scientific Publishing.

Lange, C. F. J. (2006). *Improving the quality of UML models in practice.* Paper presented at the 28th international conference on Software engineering (ICSE), Shanghai, China, pages 993-996.

Lange, C. F. J., & Chaudron, M. R. V. (2006). *Effects of defects in UML models: an experimental investigation.* Paper presented at the Proceeding of the 28th international conference on Software engineering, Shanghai, China, May 20-28, 2006, pages 401-411.

Lange, C. F. J., Chaudron, M. R. V., & Muskens, J. (2006). In practice: UML software architecture and design description. *Software, IEEE, 23*(2), 40-46.

Laplante, P. A., & Neill, C. J. (2006). *Antipatterns: Identification, Refactoring, and Management* (1st Edition). Roca Baton: Auerbach (Taylor & Francis Group).

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

Liu, W. (2002). *Rule-Based Detection Of Inconsistency In Software Design.* Master Thesis, University of Toronto, Toronto, Canada, from http://www.cs.toronto.edu/fm/pubs/pdf/liu02b.pdf.

Liu, W., Easterbrook, S., & Mylopoulos, J. (2002). *Rule-Based Detection Of Inconsistency In Uml Models.* Paper presented at the Workshop on Consistency Problems in UML-Based Software Development (WCPUSD) at the Fifth International Conference on the Unified Modeling Language (UML), Dresden, Germany, October 20, 2003, pages 106-123.

Livshits, B. V., & Lam, M. S. (2005). *Finding security vulnerabilities in java applications with static analysis.* Paper presented at the Proceedings of the 14th conference on USENIX Security Symposium - Volume 14, Baltimore, MD, pages 271-286.

Long, J. (2001). Software reuse antipatterns. *Software Engineering Notes (SEN), 26*(4).

Longshaw, A., & Woods, E. (2004). *Patterns for Generation, Handling and Management of Errors.* Paper presented at the OT, pages 26.

Longshaw, A., & Woods, E. (2005). *More Patterns for the Generation, Handling and Management of Errors.* Paper presented at the EuroPLOP, pages 14.

Lorentz, M., & Kidd, J. (1994). *Object-Oriented Software Metrics: a Practical Guide*: Prentice Hall.

Love, T. (1991). Timeless Design of Information Systems. *Object Magazine, November-December 1991*, 46.

Mäntylä, M. (2003). *Bad Smells in Software - a Taxonomy and an Empirical Study*. Master Thesis, University of Technology, Helsinki, from http://www.soberit.hut.fi/sems/shared/deliverables_public/mmantyla_thesis_final.pdf.

Mäntylä, M., Vanhanen, J., & Lassenius, C. (2003). *A taxonomy and an initial empirical study of bad smells in code.* Paper presented at the International Conference on Software Maintenance (ICSM), Amsterdam Netherlands, 22-26 Sept. 2003, pages 381-384.

Marinescu, R. (2002). *Measurement and Quality in Object-Oriented Design.* PhD Thesis, Politehnica University of Timisoara, Timisoara.

Marinescu, R., & Lanza, M. (2006). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems* (1st Edition): Springer.

Martin, R. C. (2000). *Design Principles and Design Patterns*: ObjectMentor.

Mellor, S. J., Kendall, S., Uhl, A., & Weise, D. (2004). *MDA Distilled*: Addison Wesley Longman Publishing Co., Inc.

Melton, H., & Tempero, E. (2006). *Identifying Refactoring Opportunities by Identifying Dependency Cycles.* Paper presented at the Twenty-Ninth Australasian Computer Science Conference (ACSC), Hobart, TAS, Australia, January 16 - 19, 2006, pages 35 - 41

Mendes, E. (2005). *A systematic review of Web Engineering Research.* Paper presented at the International Symposium on Empirical Software Engineering, pages 498-507.

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

Moha, N., & Guéhéneuc, Y.-G. (2005). *On the Automatic Detection and Correction of Software Architectural Defects in Object-Oriented Designs.* Paper presented at the 6th International Workshop on Object-Oriented Reengineering (WOOR) in conjunction with the 19th European Conference on Object-Oriented Programming (ECOOP), July 2005, pages

Moha, N., Huynh, D.-L., & Gueheneuc, Y. G. (2005). *A Taxonomy and a First Study of Design Pattern Defects.* Paper presented at the International Workshop on Design Pattern Theory and Practice (IWDPTP), Budapest, Hungary, September 25-30, pages

Monteiro, M. P., & Fernandes, J. M. (2006). Towards a Catalogue of Refactorings and Code Smells for AspectJ. *Transactions on Aspect-Oriented Software Development (TAOSD)*, 214-258.

Munro, M. J. (2005). *A Measurement-Based Approach for Detecting Design Problems in Object-Oriented Systems* (No. EFoCS-57-2005).

Nakamura, T. (2007). HPC Bug Base. Retrieved 16. June, 2007, from http://www.hpcbugbase.org

Nkwocha, F., & Elbaum, S. (2005). *Fault patterns in Matlab.* Paper presented at the First workshop on End-user software engineering (WEUSE) in conjunction with the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, pages 1-4.

Ortega, M., Perez, M., & Rojas, T. (2003). Construction of a Systemic Quality Model for Evaluating a Software Product. *Software Quality Journal, 11*(3), 219-242.

Pai, M., McCulloch, M., Gorman, J. D., Pai, N., Enanoria, W., Kennedy, G., et al. (2004). Systematic Literature and Meta-Analyses: An illustrated, step-by-step guide. *National Medical Journal of India, 17*(2), 86-95.

Parsons, T., & Murphy, J. (2004a). *Data Mining for Performance Antipatterns in Component Based Systems Using Run-Time and Static Analysis.* Paper presented at the 6th International Conference on Technical Informatics (CONTI), Timisoara, Romania, May 2004, pages 113-118.

Parsons, T., & Murphy, J. (2004b). *A Framework for Automatically Detecting and Assessing Performance Antipatterns in Component Based Systems using Run-Time Analysis.* Paper presented at the 9th International Workshop on Component Oriented Programming (WCOP), in conjunction with 18th European Conference on Object-Oriented Programming (ECOOP), June 2004, pages 8.

Perry, W. E. (2000). *Effective Methods of Software Testing, Second Edition*: John Wiley & Sons Inc.

Petroni, N. L., Jr., & Arbaugh, W. A. (2003). The dangers of mitigating security design flaws: a wireless case study. *IEEE Security & Privacy Magazine (ISPM), 1*(1), 28-36.

PPR. (2005). Portland Pattern Repository. Retrieved 10. Oct., 2005, from http://c2.com/ppr/, http://en.wikipedia.org/wiki/Portland_Pattern_Repository

Rech, J. (2004). *Towards Knowledge Discovery in Software Repositories to Support Refactoring.* Paper presented at the Workshop on Knowledge Oriented Maintenance (KOM) at SEKE 2004, Banff, Canada, pages 462-465.

Rech, J., & Ras, E. (2007, in work). Aggregation von Erfahrungen in Erfahrungsdatenbanken. *Künstliche Intelligenz*, 6.

Rech, J., Ras, E., & Decker, B. (2007). Intelligent Assistance in German Software Development: A Survey. *IEEE Software, 24*(4), 72-79.

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

Riel, A. J. (1996b). *Object-Oriented Design Heuristics* (1st Edition): Addison-Wesley Professional.

Rising, L. (2000). *The pattern almanac 2000*. Boston: Addison-Wesley.

Robbins, J. E. (1998). *Design Critiquing Systems* (No. Tech Report UCI-98-41).

Robbins, J. E. (1999). *Cognitive Support Features for Software Development Tools.* Ph.D. Thesis, University of California, Irvine.

Robbins, J. E., Hilbert, D. M., & Redmiles, D. F. (1997). Argo: a design environment for evolving software architectures. *Proceedings of International Conference on Software Engineering. ICSE 97, Boston, MA, USA, 17 23 May 1997 * New York, NY, USA: ACM, 1997, p 600 1.*

Robbins, J. E., Hilbert, D. M., & Redmiles, D. F. (1998a). Extending design environments to software architecture design. *11th Knowledge Based Software Engineering Conference, Syracuse, NY, USA, 25 28 Sept. 1996 * Automated Software Engineering, Netherlands * vol 5 (July 1998), no 3, p 261 90, 58 refs.*

Robbins, J. E., Hilbert, D. M., & Redmiles, D. F. (1998b). Software architecture critics in Argo. *Proceedings of 1998 International Conference on Intelligent User Interfaces, San Francisco, CA, USA, 6 9 Jan. 1998 * New York, NY, USA: ACM, 1998, p 141 4.*

Robbins, J. E., Medvidovic, N., Redmiles, D. F., & Rosenblum, D. S. (1998c). Integrating architecture description languages with a standard design method. *Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan, 19 25 April 1998 * Los Alamitos, CA, USA: IEEE Comput. Soc, 1998, p 209 18.*

Robbins, J. E., & Redmiles, D. F. (1998). Software architecture critics in the Argo design environment. *Knowledge Based Systems, 11*(1), 47-60.

Robbins, J. E., & Redmiles, D. F. (2000). Cognitive support, UML adherence, and XMI interchange in Argo/UML. *Information and Software Technology, Netherlands * vol 42 (25 Jan. 2000), no 2, p 79 89, 25 refs.*

Roock, S., & Lippert, M. (2006). *Refactoring in Large Software Projects* (Paperback): John Wiley & Sons.

SAP. (2005a). Secure Programming - ABAP. from https://www.sdn.sap.com/irj/sdn/devguide2004s

SAP. (2005b). Secure Programming - Java. from https://www.sdn.sap.com/irj/sdn/devguide2004s

SAP. (2007a). Enterprise Service-Oriented Architecture (Enterprise SOA). from http://www.sap.com/platform/esoa/index.epx

SAP. (2007b). SAP CRM. from http://www.sap.com/solutions/business-suite/crm/index.epx

Schmidmeier, A. (2004). *Patterns and an Antiidiom for Aspect Oriented Programming.* Paper presented at the EuroPLoP, pages 21.

SEI. (2006). *CMMI for Development, Version 1.2* (No. CMU/SEI-2006-TR-008).

Shadrin, G. (2005). Three Sources of a Solid Object-Oriented Design - Design heuristics, scientifically proven OO design guidelines, and the world beyond the beginning. *JAVA developer's journal (JDJ), 10*(5).

Simon, F., Olaf Seng, O., & Mohaupt, T. (2006). *Code Quality Management* (1st Edition): Dpunkt Verlag.

Simon, F., Steinbruckner, F., & Lewerentz, C. (2001). *Metrics based refactoring.* Paper presented at the 5th European Conference on Software Maintenance and Reengineering (CSMR), Lisbon Portugal, 14-16 March 2001, pages 30-38.

Smith, C. U., & Williams, L. G. (2001). *Software Performance AntiPatterns - Common Performance Problems and their Solutions.* Paper presented at the 27th International Computer Measurement Group Conference (ICMG), pages 797-806.

Smith, C. U., & Williams, L. G. (2002). *New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot.* Paper presented at the 28th International Computer Measurement Group Conference (ICMG), Reno, Nevada, USA, December 8-13, 2002, pages 667-674.

Smith, C. U., & Williams, L. G. (2003). *More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot.* Paper presented at the 29th International Computer Measurement Group Conference (ICMG), pages 717-725.

Stewart, D. B. (1999). 30 pitfalls for real-time software developers. *Embedded Systems Programming (ESP), 12*(11).

Stürmer, F. (2006). *A Rule-based Approach for Business Logic Modelling in CRM Business Objects.* Diploma Thesis, University of Mannheim & SAP-AG.

Sutter, H., & Alexandrescu, A. (2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* (1st Edition (October 25, 2004)): Addison-Wesley Professional.

Tahvildari, L., Kontogiannis, K., & Mylopoulos, J. (2003). Quality-driven software re-engineering. *Journal of Systems and Software, 66*(3), 225-239.

Tate, B. (2002). *Bitter Java* (1st Edition (April 2002)): Manning Publications Co.

Tate, B., Clark, M., Lee, B., & Linskey, P. (2003). *Bitter EJB* (1st Edition): Manning Publications Co.

Taylor, R. N., & Osterweil, L. J. (1980). Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering (TSE), 6*(3), 265-278.

Telles, M. A., & Hsieh, Y. (2001). *The Science of Debugging* (1st Edition): Coriolis Group Books.

TopCased. (2007). TopCased. from http://www.topcased.org/

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.

Tourwé, T., & Mens, T. (2003). *Identifying refactoring opportunities using logic meta programming.* Paper presented at the Seventh European Conference on Software Maintenance and Reengineering (CSMR), Benevento Italy, 26-28 March 2003, pages 91-100.

van Emden, E., & Moonen, L. (2002). Java quality assurance by detecting code smells. *Reengineering Forum; Virginia Commonwealth Univ.; IEEE Comput*, Burd, E.. - Los Alamitos, CA, USA, USA IEEE Comput. Soc, 2002, x+2349 2097-2106, 2025 Refs.

Vermeulen, A., Ambler, S. W., Bumgardner, G., Metz, E., Misfeldt, T., Shur, J., et al. (2000). *The Elements of Java Style* (1st Edition): Cambridge University Press.

Veryard, R. (2001). Design pitfalls as negative patterns. Retrieved 1. June, 2007, from http://www.users.globalnet.co.uk/~rxv/sqm/pitfalls.htm

Vide, P. (2007a). *Project Deliverable D1.1*: VIDE Project.

Vide, P. (2007b). *Project Deliverable D3.1*: VIDE Project.

Visaggio, G. (2001). Ageing of a data-intensive legacy system: symptoms and remedies. *Journal of Software Maintenance and Evolution, 13*(5), 281-308.

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

Waters, R. C. (1994). Cliché-based program editors. *ACM Trans. Program. Lang. Syst., 16*(1), 102-150.

Webster, B. F. (1995). *Pitfalls of object-oriented development* (1st Edition): M & T Books.

White, A., & Schmidt, K. (2005). Systematic literature reviews. *Complementary Therapies in Medicine, 13*(1), 54-60.

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

Wikipedia. (2007). Coding conventions for languages. from http://en.wikipedia.org/wiki/Programming_style#Coding_conventions_for_languages

Wohlin, C., Aurum, A., Petersson, H., Shull, F., & Ciolkowski, M. (2002). Software inspection benchmarking-a qualitative and quantitative comparative opportunity. *Proceedings Eighth IEEE Symposium on Software Metrics, Ottawa, Ont., Canada, 4 7 June 2002 * Los Alamitos, CA, USA: IEEE Comput. Soc, 2002, p 118 27.*

Wooldridge, M. J., & Jennings, N. R. (1999). Software engineering with agents: pitfalls and pratfalls. *IEEE Internet Computing (IIC), 3*(3), 20-27.

Younessi, H. (2002). *Object-Oriented Defect Management of Software* (1st Edition): Prentice Hall PTR.

Yourdon, E. (1993). *Object-Oriented Systems Design: An Integrated Approach* (1st Edition): Prentice Hall.