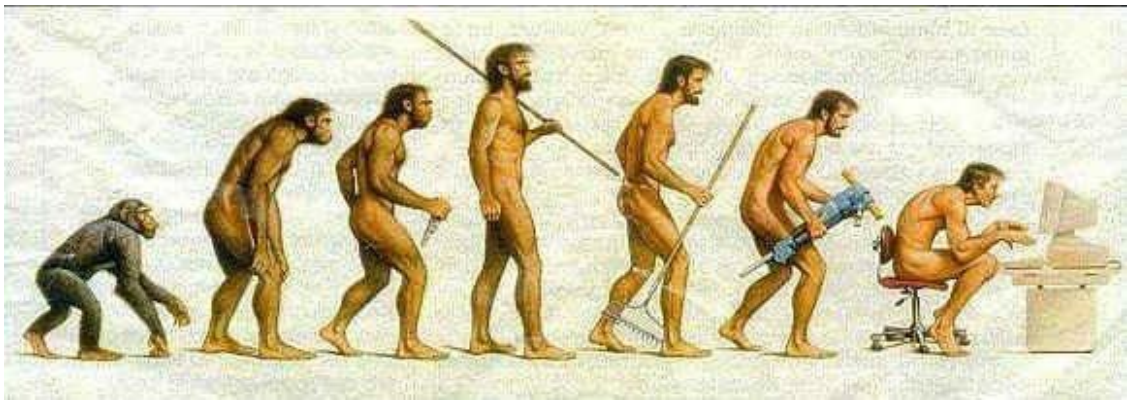
ITEA
INFORMATION TECHNOLOGY
FOR EUROPEAN ADVANCEMENT

SERIOUS

DELIVERABLE
D3.3 – Overview and evaluation of design and refactoring methods



Project number:	ITEA 04032
Document version no.:	WP3 Deliverable 3.3 Final version
Edited by:	University of Antwerp “January 14, 2006”

ITEA Roadmap domains:
Major: Services & software creation

ITEA Roadmap categories:
Major: Software engineering
Minor: System engineering

HISTORY

Document version #	Date	Remarks
V0.1	14-10-2006	Initial integration of questionnaire results, University of Antwerp
V0.2	19-12-2006	Addition of FMEA method (section 2.3) by Philips PMS
V1.0	14-01-2006	Final version

TABLE OF CONTENTS

1 Executive summary

This document presents a description of the state-of-practice in the application of best practices in software design and software refactoring. The objective of this endeavour is the documentation of design and refactoring methods that are used within the consortium, and which improve quality. Thus, a means for collecting best practices was required.

To gather a set of best practices applied within the consortium, an online questionnaire was composed. This questionnaire presented examples of best practices in quality-oriented design and refactoring, e.g.:

- Two examples of a design process:
  - Rapid prototyping
  - Design by contract
- An example of an anti-pattern:
  - The round-tripping performance anti-pattern

The questionnaire enabled the description of the following types of best practices:

- design patterns (good examples), or conversely, anti-patterns (counter examples). The focus can be both on a before-and-after comparison, and on the steps required to get there.
- process guidelines, e.g., ways of working (such as early prototyping, test-driven development, design-by-contract) and key activities (such as use of Class-Collaboration-Cards)

The questionnaire was send out to the responsibles of the different partners contributing to Workpackage 3 on October 16th, 2006, and was available for collecting best practices until November, 8th, 2006.

In total, 4 best practices were gathered, which are discussed in the following sections. Section 2 discusses three *design* best practices. The single *refactoring* best practice is discussed in Section 3. Finally, a discussion on the collected best practices representing the overview and evaluation of existing design and refactoring methods is provided in Section 4.

2 Catalogue of design methods

This chapter describes the quality-oriented design methods gathered across the Workpackage 3 partners.

2.1 Diversity management by encapsulation

"Diversity management by encapsulation" represents a design choice, which introduces proxy classes to encapsulate diversity of interface semantics.

The problem arises when a server interfaces with multiple clients that need changes. Typically, the interface of these clients must remain stable, due to interactions with other servers. Moreover, client-specific details cannot be incorporated in the server, as this would disable transparent client services in the server.

This design method proposes to solve the problem by introducing *proxy classes*, which provide specific interfaces to the specific client application. Such proxy classes translate the generic client interface to the specific client interface.

The motivation for this solution is that the introduction of new clients will not require changes to the server, and can be merely accommodated through the introduction of new proxies.

2.1.1 Context

This is an architectural decision that must be made in the inception/elaboration phase, i.e., before implementation starts.

2.1.2 Actions

Define a *generic client interface*, which is applicable for all clients. For each client, implement a proxy class, which translates the generic client interface to the specific client interface.

2.1.3 Quality trade-offs

Since the proxy class encapsulates the interface translation, maintainability (in particular extendibility) is positively affected. Performance can be slightly negatively affected in case the interface translation is computation intensive, or introduces a communication overhead.

2.1.4 Lessons learned

This practice was successfully applied in PMS in the Phoenix project, and supported the integration of a total of seven different clinical applications; five of them were third party applications that could not be changed.

2.2 Unit testing

"Unit testing" represents a design choice for test code, which refines the granularity of tests to a single unit, typically a class in the Object-Oriented paradigm.

The problem arises when it is hard to perform root-cause-analysis of faults. Typically, in these cases, the tests verify the accuracy of a whole scenario being executed, typically in the form of an integration test.

This test design method proposes to solve the problem by verifying the execution at a very fine-grained granularity, in such a manner that merely a single class can be responsible for the fault. Mostly, the interaction between classes should be addressed by introducing stubs, which mimic the behaviour of a class with which the class under test is interacting.

The motivation for this solution is that early feedback can be provided when changing units (e.g., classes) by running unit tests. Thus, an iterative style of running tests is enabled, e.g., first unit testing, then integration testing and subsequently system testing. Unit tests are thus highly useful for phases in which changes are applied frequently, e.g., development of new code, implementation of change requests, and refactoring.

2.2.1 Context

This is a design decision that has considerable impact on the design of classes. While it is possible to modify the structure of your system to enable unit tests, the initial design of your system has a large impact on the costs to introduce unit tests.

2.2.2 Actions

Unit testing is conducted in an automated environment, mostly through the use of a third party supplied component or framework, for testing a unit in isolation.

2.2.3 Quality trade-offs

Attempting to test a unit in isolation forces one to evaluate the unit's dependencies. As unit tests are infeasible for units with many dependencies, unit testing stimulates one to write isolated units, which are decoupled and highly cohesive. Thus, unit testing positively affects maintainability.

Moreover, by concentrating test effort on a fine-grained level, the root cause of faults can be more easily verified, thereby improving reliability.

However, the amount of code that has to be written for the unit tests and their stubs is considerable. This test code has to co-evolve with the production code, and therefore introduces an additional maintenance cost.

2.2.4 Lessons learned

Unit testing is particularly supportive for refactoring. In essence, refactoring is a controlled manner to apply a series of source modifications. During such changes, one has to continuously ensure that the behaviour of the system has not changed. In case the behaviour has changed, unit tests can direct the developer/tester to the particular unit that does not longer behave according to the original specifications.

Unit testing can stimulate a different approach to software development, i.e., Test-Driven Development (TDD). In TDD, tests are written before the actual code, thereby clarifying the requirements in a verifiable manner. Consequently, the unit's specification (and usage documentation) is provided in the unit test itself.

The fact that unit testing enforces one to write encapsulated and isolated units can also be a drawback, as this encapsulation introduces a series of abstractions. For relatively small applications, this lead to *over-engineering*.

2.3 Software design review using Failure Mode and Effect Analysis

Failure Mode and Effect Analysis (or FMEA) is a well-known design review method used to identify potential failure modes, their effects and root causes. The method offers the possibility to rank the importance of the various failure modes using the so-called Risk Priority Number, which is a multiplication of the severity of the effect, the probability of occurrence of the root causes and the delectability in the current design. This method can also be applied to software systems. The basis for this approach was described by Maxon & Olszewski [1].

Software systems fail mainly for two reasons: logic errors in the code, and exception failures. Exception failures can account for up to 2/3 of the system crashes. Traditional approaches to reducing exception failures, such as code reviews, walkthroughs and formal testing, while very useful, are limited in their ability to address a core problem: the programmer's inadequate coverage of exceptional conditions. The problem of coverage is rooted in cognitive factors that impede the mental generation (or recollection) of exception cases that would pertain in a particular situation, resulting in insufficient software robustness.

FMEA offers a structured way to consider potential exceptions (or failure modes). To do this:

- An overview is made of the various software modules that make up the software system under consideration
- Process flows are made for the various use cases (or user scenario's) in which the interaction between the various modules is depicted and described (e.g. the start-up of the system)
- For each scenario/flow possible exceptions are generated using the CHILDREN mnemonic.
- These exceptions are considered as failure modes in the FMEA framework. For each exception the possible effects and root causes are identified. Current controls are investigated and design improvements are proposed.

The CHILDREN mnemonic is shown in a typical fishbone graph displayed in Figure 1.

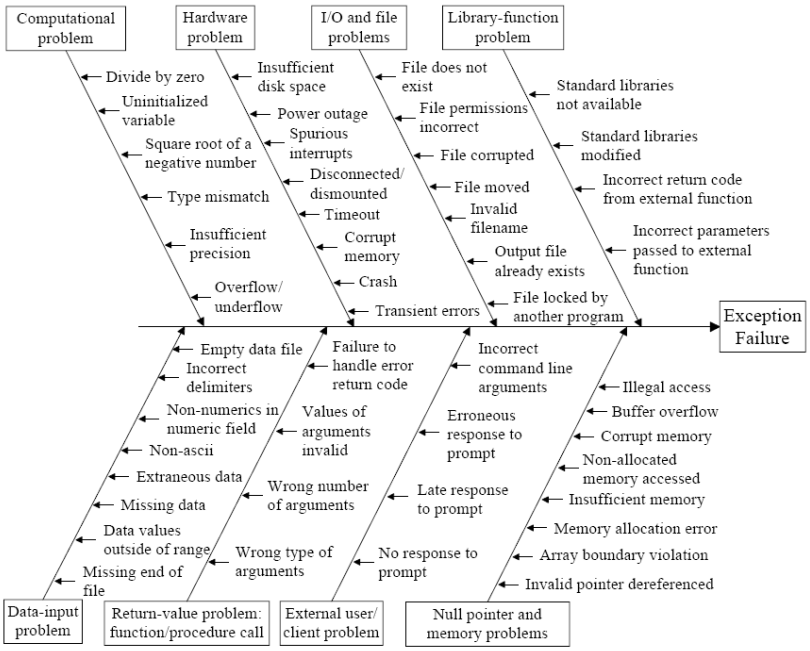


Figure 1: Fishbone diagram showing exception types and exemplars. The first letters of the rib labels spell the mnemonic CHILDREN.

2.3.1 Context

The FMEA method can be applied on all levels of the design, from system level down to the smallest software component/module. It is important to well define the scope that is considered for the FMEA. If required, separate FMEA's can be planned for lower levels in the system.

2.3.2 Actions

The various steps in the FMEA were described above. Preparation for the FMEA workshop is very important. It is also important to invite various people in the workshop. Not only designers should participate but also (if possible) users, service engineers, suppliers, etc.

2.3.3 Quality trade-offs

In principle the FMEA method can be used to evaluate various quality attributes of the software system, not only robustness. Of course the FMEA method in it self does not make the trade offs between the various quality attributes. Other methods (like the Pugh selection method) must be used for that.

2.3.4 Lessons learned

This FMEA method for optimising the robustness of software systems was successfully applied in Philips Medical Systems. Various software systems were improved and designers found the method a useful addition to other review methods.

3 Catalogue of refactoring methods

This section reports on the single refactoring method that was reported using the questionnaire.

3.1 Type conversion in legacy code

"Type conversion in legacy code" is a refactoring practice, and can be applied in the phases starting from implementation. Two examples of types between which one can converse are (i) chars; and (ii) unicode.

The problem arises when the information capacity of one type can no longer be met at in at least one part of the software system. Due to dependencies among modules, changing these types will not only have a local effect, but will also ripple to modules that interact with the data of that type.

This refactoring practice proposes to solve the problem by iteratively converting the modules. Ensure that the system remains in a working state throughout the conversion, as well as open to other changes, by carefully examining change ripples.

The motivation for this iterative solution is to control risks. A big bang approach in which all type usage would be changed would block any other changes in the meantime, and would introduce the danger that the system cannot be brought to a working state.

3.1.1 Actions

Declare new data types that encapsulate the new type, e.g., unicode, while keeping the old (non-unicode) data types. Then, iteratively (e.g., on a module basis) convert modules to make use of the new (unicode) data types. After each module change, test the module in isolation, and rerun the integration tests.

New development should be required to use the new (unicode) data types, and be unicode-compliant itself.

3.1.2 Quality trade-offs

Since the new data types encapsulate the choice between a char and a unicode, future changes to the type will not spread through the system. Accordingly, maintainability is positively affected.

3.1.3 Lessons learned

This practice relies extensively on testing practices. Well-defined unit tests are necessary to detect and pinpoint conversion problems as soon as possible.

4 Discussion

The collection of best practices using an online questionnaire provided an instrument to verify the extent to which design and refactoring methods that improve quality are incorporated in the different industrial partners contributing in the project. This instrument was set up to derive a representative reflection, and consequently, we can assume that the overview presented in this document reflects the actual state-of-practice with regard to quality-oriented design and refactoring in the consortium.

The use of the questionnaire resulted in the documentation of three best practices: 3 design methods and a single refactoring method. Interestingly, the three discussed best practices stimulate encapsulation to improve maintainability. One design method focuses on reliability (or robustness), but could as well be applied to other quality characteristics.

Thus, these best practices acknowledge that software design can indeed be steered towards quality-optimizations (in particular maintainability and reliability), and provide illustrative examples of how such optimizations can be achieved.

However, the fact that merely 4 best practices were reported seems to indicate that the concept of quality-oriented design methods or quality-oriented refactoring methods is not extensively put in practice yet.

Consequently, the main conclusion of this effort is that there is considerable room for improvement in the area of quality-oriented design and refactoring.

5 References

(1) Roy A. Maxon and Robert T. Olszewski. *Improving Software Robustness with Dependability Cases*. 28th International Symposium on Fault-Tolerant Computing: Munich, Germany, 23-25 June 1998