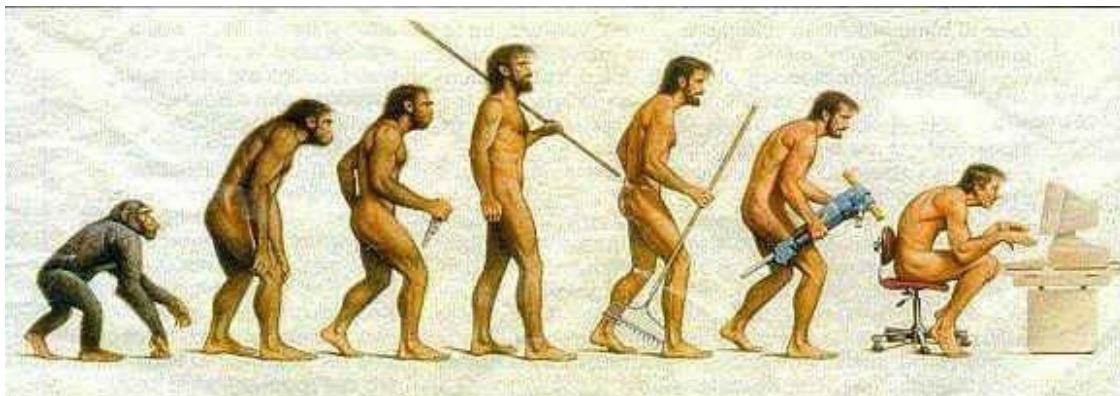




# SERIOUS

## DELIVERABLE

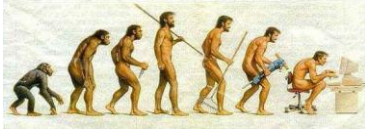
### D3.7 – Relationships among metrics, models and designs



Project number: ITEA 04032  
Document version no.: WP3 Deliverable 3.7 Final version  
Edited by: UPM, UA, UH

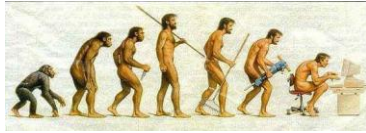
**ITEA Roadmap domains:**  
Major: Services & software creation

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



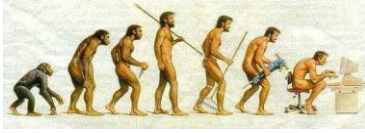
## HISTORY

Document version #	Date	Remarks
V0.1	November 23 <sup>rd</sup> , 2007	Starting version, template definition of ToC.
V0.2	November 30 <sup>th</sup> , 2007	Rough draft: contributions from all partners.
V0.3	November 29 <sup>th</sup> , 2007	First reviews.
V0.4	December 10 <sup>th</sup> , 2007	Second reviews integrated.
V0.5	December 14 <sup>th</sup> , 2007	Final draft to sign of by PCC members
V1.0		Final Version



## TABLE OF CONTENTS

<b>1. EXECUTIVE SUMMARY .....</b>	<b>4</b>
<b>2. INTRODUCTION .....</b>	<b>5</b>
<b>3. ASSESSING MAINTENANCE HOTSPOTS .....</b>	<b>6</b>
<b>3.1 Assessing maintainability using metrics .....</b>	<b>6</b>
<b>3.2 Test planning - detecting problematic architecture sections .....</b>	<b>18</b>
<b>3.3 Characterizing maintainability across multiple versions .....</b>	<b>22</b>
<b>4. PREDICTING MAINTAINABILITY .....</b>	<b>34</b>
<b>4.1 Explaining development productivity variation using early metrics .....</b>	<b>34</b>
<b>4.2 Release planning - estimating resource cost of CR and new reqs .....</b>	<b>42</b>
<b>5. SUMMARY .....</b>	<b>45</b>
<b>5.1 Assessing maintainability using metrics .....</b>	<b>45</b>
<b>5.2 Test planning - detecting problematic architecture sections .....</b>	<b>45</b>
<b>5.3 Characterizing maintainability across multiple versions .....</b>	<b>45</b>
<b>5.4 Explaining development productivity variation using early metrics .....</b>	<b>46</b>
<b>5.5 Release planning - estimating resource cost of CR and new reqs .....</b>	<b>46</b>
<b>6. REFERENCES .....</b>	<b>47</b>



## 1. Executive Summary

Engineers facing the challenge of software evolution are confronted with systems whose design no longer facilitates upcoming feature requests. Accordingly, the need arises to quantify design quality, and to identify design weaknesses whose improvement will speed up future extensions.

Through an overview of industrial case studies, this document illustrates how maintainability can be managed proactively in practice. The case studies share the goal of measuring and/or predicting maintainability through the application of a quality model.

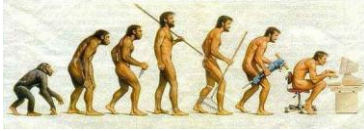
A first case study extends the ISO 9126 quality standard with existing internal metrics. This quality model is applied to evaluate whether a migration to a service-oriented architecture improved maintainability.

Secondly, a tool is introduced that supports the detection of design patterns. The tool is used to detect the affected sections of a reusable interface when making modifications to a framework. This information can be used to plan tests for the interface functionality.

The final case study on measuring maintainability demonstrates how visualizations can support quality assurance. These visualizations present how the value of relevant maintainability indicators evolves across multiple versions, either at the software module level, or at the individual class level.

The second part of this deliverable elaborates how maintainability can be predicted. The first case study addresses the intrinsic difficulty in estimations, namely variation. A baseline model for predicting development effort is enriched with an additional factor, resulting in increased prediction accuracy.

The last case study discusses the problem of calculating the size of a framework reuse interface. The size can be used to estimate the effort of modifying or refactoring the interface.



## 2. Introduction

Software maintenance issues primarily dominate costs in software development. If in the seventies, estimation studies claimed that maintenance consumed 67% of total software costs [ZSG79], nowadays some authors are already talking about 90% [Erl00]. Other studies have shown that approximately 50% of the time is spent understanding the code [Sta84]. Solving bugs, improving performance, applying security patches or adding new features are part of the everyday jobs of a software developer. The ISO 9126 [ISO9126] standard defines maintainability as the capability of the software product to be modified, including corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

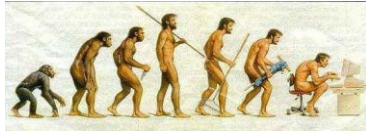
In terms of software quality, we can distinguish internal and external quality [ISO9126]:

- Internal quality is the totality of characteristics of the software product from an internal view, that is, the code. Details of software product quality can be improved during code implementation, reviewing and testing.
- External quality is the totality of characteristics of the software product from an external view. It is the quality when the software is executed, which is typically measured and evaluated while testing in a simulated environment with simulated data using external metrics.

This document illustrates several case studies concerning maintainability across internal and external quality. These case studies are the following:

- Maintainability across internal quality (section 3):
  - Section 3.1 presents a method to estimate maintainability, using internal quality indicators (metrics).
  - Section 3.2 introduces Maisa (Metrics for Analysis and Improvement of Software Architectures). It is a tool for architecture-based measurement and evaluation of software quality.
  - Section 3.3 presents a method to predict changeability and testability using Chidamber and Kemerer object-oriented metrics.
- Maintainability across external quality (section 4).
  - Section 4.1 presents a study about the estimation of development effort as a productivity indicator.
  - Section 4.2 introduces how estimating the cost of future releases in software product lines.

These case studies perform the body of the present document, which is summarized Section 5.



## 3. Assessing maintenance hotspots

This chapter presents three case studies on evaluating maintainability. To bridge the gap between quality evaluations, and its improvement, the case studies focus on the detection of so called *hotspots*.

Maintainability hotspots are parts of a software system that are suspected of introducing an extraordinary burden to the maintainer. Evidently, this suspicion is not a proof that maintainability is indeed a problem, yet it provides a starting point for discussions with the maintainers.

In case these maintainers indeed recognize the software part as problematic with regard to maintainability (e.g., it is undergoing frequent changes, and its design is not up to date to these changes), management might decide to address this maintainability issue.

### 3.1 Assessing maintainability using metrics

#### 3.1.1 Introduction

ISO 9126 identified six main external quality attributes: functionality, reliability, usability, maintainability, portability and efficiency [ISO9126]. Maintainability is the level of effort required for modifying the software product. ISO 9126 divides maintainability in four main characteristics: analysability, changeability, stability and testability [ISO9126].

External attributes are the most important indicators of final product quality, but they can only be effectively measured after the product has been developed.

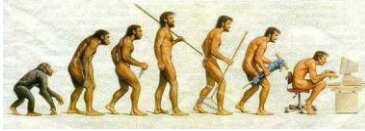
Other approach for measuring the quality of a software system is through internal quality attributes. Some examples of these attributes are coupling, complexity, cohesion and so on. Internal attributes allows the prediction of software quality early in the development life-cycle [Fen94].

It has been empirically shown in previous studies that design level coupling and cohesion in isolation can have a causal impact on external quality attributes such as maintainability or fault-proneness in both procedural [HS90] and object oriented software [AW03].

In this study, we propose a method to predict the evolution of the maintainability of a software system using internal quality indicators, that is, structural metrics. The underlying assumption by the keeping good levels of internal quality, the external quality will achieve good levels too. This way, developers can be responsible of the expected external quality of the software system by improving internal quality in development time.

#### 3.1.2 Relation between external and internal attributes

The first step in this study is the establishment of a relationship between maintainability (external quality) and internal quality indicators. There are some examples in the literature of this kind of relation.



Ying Zou proposes a maintainability soft-goal dependency graph. This kind of graph is used to make this association of features and software qualities explicit [ZK02].

Marinescu and Ratiu proposed the Factor-Strategy Model (FSM) that quantifies a systems design conformance to a set of design principles, rules and heuristics [MR04]. The FSM uses the detection strategies to identify design problems that can be connected with quality attributes of a systems design [Mar02].

Following the ideas proposed by Zou and Marinescu, we present a new graph which establishes the relation from maintainability to internal quality attributes. The first step is divide maintainability according to ISO 9126, that is, analysability, changeability, stability and testability. We have composed this graph by composition of features. For example, maintainability is the result of adding analysability, changeability, stability and testability. In the next level, analysability will be composed by the aggregation of internal quality attributes. We have selected a group of internal quality attributes [Fdc07]:

- **Complexity** is the measure of how difficult a software system is. It is desirable to achieve low complexity in software systems.
- **Coupling** is the degree to which each program module depends on the others. It is desirable to achieve low coupling in software systems.
- **Cohesion** is defined as “*a measure of the degree to which the elements of a module belong together*” [SMY74]. It is desirable to achieve high cohesion in software systems.
- **Change Impact** is the measure of the effect that supposes changing some part in software systems. It is desirable to achieve low change impact.
- **Test Coverage** is the amount of checks performed in a software system. It is desirable to achieve high test coverage.
- **Code Quality** is the measure of the code form. A good programmer must follow code conventions (indentation, normalization). Besides of that, the code must be well commented. Therefore, high code quality is desirable.
- **Encapsulation** in object-oriented programming is the technique of keeping together data structures and the methods which act on them. High encapsulation is desirable.
- **Reuse** is the amount of code developed for one application program that is used in another application. High reuse is desirable.

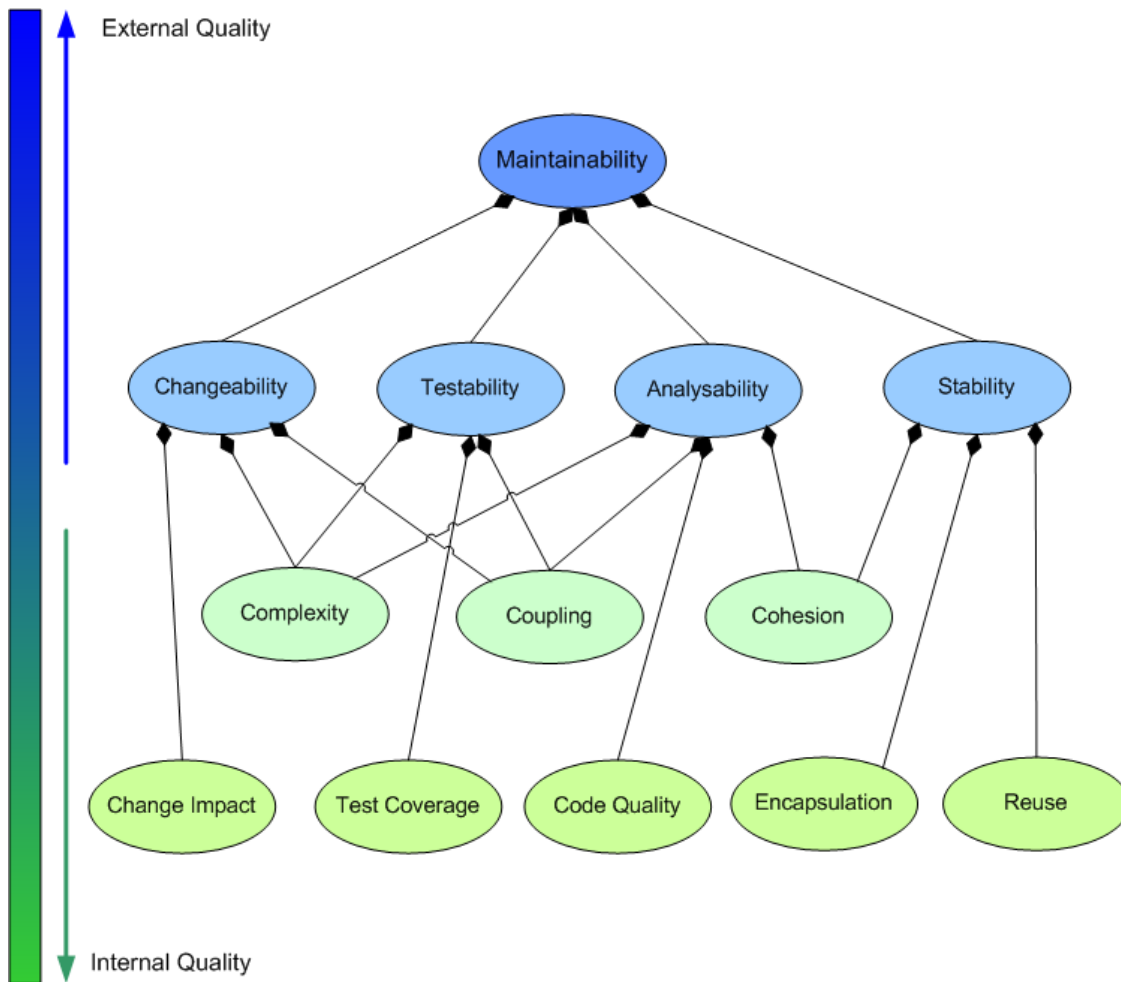
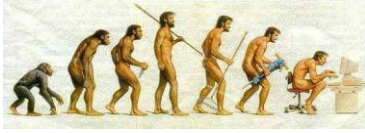
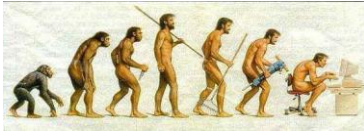


Figure 1 - Relationship between maintainability and internal quality

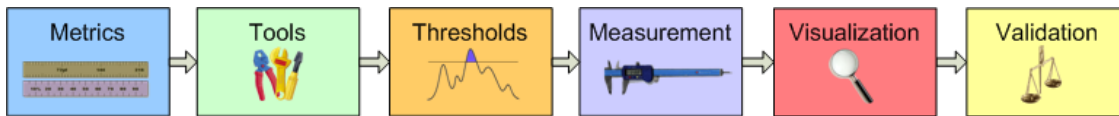
### 3.1.3 Proposed Method

We have now reached the start point of this study. Figure 1 shows the relation between maintainability and internal quality indicators. At this point, we continue with the following steps:

- *Selecting metrics.* It is important to select a set of metrics for each one of the internal quality indicators (complexity, coupling, cohesion, and so on).
- *Selecting tools.* For each metric, we should have the appropriate tool for taking the value.
- *Calculating thresholds.* This is a critical and difficult point. The world of metrics is very heterogonous. In this step we try to estimate the optimum range of operation for each metric selected.
- *Measurement.* Taking the selected metrics with tools.
- *Visualization.* We have to choose a visualization technique for helping us to better understand the evaluation of the results.



- *Validation.* With the collected information, in this step it's time to evaluate and interpret the results.



**Figure 2 - Method to evaluate maintainability**

In the next sections we inquire into those steps.

### 3.1.3.1 Metrics

The first step is selecting a group of metric for each one of the internal quality attributes. These attributes correspond to green ellipses in Figure 2: complexity, coupling, cohesion, change impact, test coverage, code quality, encapsulation and reuse.

The choice and interpretation of metrics is full of problems [KB04]. There are a lot of metrics in the literature. Since the perfect software metric does not exist, we have selected a set of metrics, those with more impact:

- Software Complexity Metrics by McCabe, 1976 [McC76].
- Software Complexity Metrics by Halstead, 1977 [Hal97].
- Object-Oriented Metrics by Chidamber and Kemerer, 1994 [CK94].
- Object-Oriented Design Quality Metrics by Martin, 1994 [Mar94].
- Object-Oriented Design by Abreu, 1994 [AC94].
- Cohesion and Reuse in an Object-Oriented System, James M. Bieman and Byung-Kyoo Kang, 1995 [BK95].
- Object-Oriented Metrics by Henderson-Sellers, 1996 [Hen96].
- A Unified Framework for Cohesion Measurement in Object-Oriented Systems, by Briand, Daly, and Wüst [BDW98].

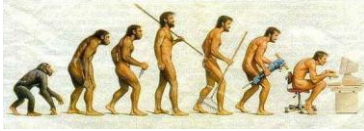
### Complexity

→ Cyclomatic Complexity [McC76] directly measures the number of linearly independent paths through a program's source code. Cyclomatic complexity is computed using a graph that describes the control flow of the program. The nodes of the graph correspond to the commands of a program. A directed edge connects two nodes if the second command might be executed immediately after the first command.

$$MCC = E - N + 2P$$

...where:

MCC = cyclomatic complexity



$E$  = the number of edges of the graph  
 $N$  = the number of nodes of the graph  
 $P$  = the number of connected components.

The minimum value for  $VG$  is 1 (one path). When  $VG$  is high, this is an indicator of high complexity, so  $VG$  must be low.

→ Program Difficulty (HD) [Hal97] estimates the level of difficulty in a piece of software.

$$HD = 1/L$$

...where:

HD = program difficulty  
L = program level.  $L = (N1/n1) * (n2/N2)$ .  
N = Program length.  $N = N1 + N2$   
n = program vocabulary.  $n = n1 + n2$   
N1 = total number of operators  
N2 = total number of operands  
n1 = count of unique operators  
n2 = count of unique operands

This value is better to be kept to a minimum.

→ Response for a Class [CK94] is the set of methods that can potentially be executed in response to a message received by an object of that class. RFC is simply the number of methods in the set.

$$RFC = M + R$$

...where:

RFC = response for a class  
M = number of methods in the class  
R = number of remote methods directly called by methods of the class

In general terms, a low value of RFC indicates greater polymorphism [MDP07].

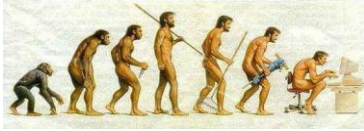
### **Coupling**

→ Efferent Coupling (ECC) [Mar94] is the total number of external classes coupled to classes of a package due to outgoing coupling (coupling to classes external classes of the package).

→ Afferent Coupling (ACC) [Mar94] the total number of external classes coupled to classes of a package due to incoming coupling (coupling from classes external classes of the package).

→ Coupling between Object Classes (CBO) [CK94] is the number of classes to which a class is coupled. High CBO is undesirable.

### **Cohesion**



→ Lack of Cohesion in Methods (LCOM) is a measure for the number of not connected method pairs in a class representing independent parts having no cohesion. It represents the difference between the number of method pairs not having instance variables in common, and the number of method pairs having common instance variables [CK94]. Given  $n$  methods  $M_1, M_2, \dots, M_n$  contained in a class  $C_1$  which also contains a set of instance variables  $\{I_i\}$ . Then for any method  $M_i$  we can define the partitioned set of  $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$  and  $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$

$$\begin{aligned} LCOM &= |P| - |Q|, \text{ if } |P| > |Q| \\ LCOM &= 0 \text{ otherwise} \end{aligned}$$

→ Lack of Cohesion in Method 5 (LCOM5) [Hen96]. LCOM has been shown to have a number of drawbacks, so it should be used with caution. LCOM5 overcomes these drawbacks. LCOM5 is normalized therefore it can be compared among classes. LCOM5 is defined as follow:

$$LCOM5 = \frac{\left( \frac{1}{a} \right) \cdot \sum_{1 \leq j \leq a} \mu(A_j) - m}{1 - m}$$

...where:

- LCOM5 = lack of cohesion in method 5
- $\{M_i\}$  ( $i = 1, \dots, m$ ) = methods accessing  $\{A_j\}$
- $\{A_j\}$  ( $j = 1, \dots, a$ ) = set of instance variables
- $\mu(A_j)$  = number of methods that reference  $A_j$

→ COH [BDW98] is a variation of LCOM5:

$$COH = \frac{\sum_{1 \leq j \leq a} \mu(A_j)}{m \cdot a}$$

**Change Impact**

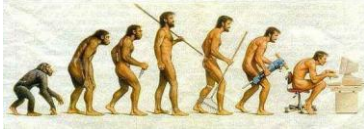
→ Weighted Methods per Class (WMC) [CK94] directly measures the method count for a class.

$$WMC = \text{number of methods defined in class}$$

According to change impact studies [CKK+02], the higher is the WMC value, the higher is the change impact.

**Test Coverage**

→ Code Accessed by Tests (COV) is the test code coverage provided by automated tests (e.g. JUnit tests, not manual QA).



$$\text{COV} = (\text{code accessed by tests LOC}) * 100 / (\text{Total LOC})$$

The ideal situation is coverage of 100%.

### **Code Quality**

→ Comments Ratio (CR) is the density of comments in the code.

$$\text{CR} = (\text{Comments LOC}) * 100 / (\text{Total LOC})$$

→ Naming Conventions (CON). With this metric we measure the number of errors in code conventions. Ideally this number should be zero, but too it is acceptable a low value.

$$\text{CON} = \text{number of errors in naming conventions}$$

The ideal situation is no errors in naming conventions (CON = 0).

### **Encapsulation**

→ Distance from the Main Sequence (DMS) [Mar94] measures the balance between the abstraction and instability rates in a package.

$$\text{DMS} = |\text{Abstraction} + \text{Instability} - 100|$$

### **Reuse**

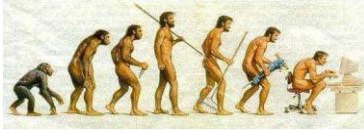
→ DIT Depth of Inheritance Tree (DIT)

$$\text{DIT} = \text{maximum inheritance path from the class to the root class}$$

→ Number of Children (NOC) is counted via the Inherits statement. It equals the number of immediate child classes derived from a class via the Inherits statement.

$$\text{NOC} = \text{number of immediate sub-classes of a class}$$

Chidamber et al. suggested that low values of DIT and NOC indicate that the reuse opportunities (via inheritance) were perhaps compromised in favour of comprehensibility of the overall architecture of the applications [CK98].



### 3.1.3.2 Tools

Our study focuses on object-oriented software systems. Concretely, the language we consider is Java. Therefore, the tools we have looked for are Java tools for taking metrics. In the next table we summarize those tools:

Tool	License	Website
Lachesis	LGPL	<a href="http://lachesis.sourceforge.net/">http://lachesis.sourceforge.net/</a>
Eclipse Metrics	CPL	<a href="http://eclipse-metrics.sourceforge.net/">http://eclipse-metrics.sourceforge.net/</a>
Metrics	CPL	<a href="http://metrics.sourceforge.net/">http://metrics.sourceforge.net/</a>
CodePro AnalytiX	Shareware	<a href="http://www.instantiations.com/codepro/analytix/about.html">http://www.instantiations.com/codepro/analytix/about.html</a>
RefactorIt	Freeware	<a href="http://www.agris.com/display/ap/RefactorIt/">http://www.agris.com/display/ap/RefactorIt/</a>
Checkstyle	LGPL	<a href="http://checkstyle.sourceforge.net">http://checkstyle.sourceforge.net</a>
Cobertura	Apache	<a href="http://cobertura.sourceforge.net/">http://cobertura.sourceforge.net/</a>
JDepend4Eclipse	BSD	<a href="http://andrei.gmxhome.de/jdepend4eclipse/">http://andrei.gmxhome.de/jdepend4eclipse/</a>
JFamix	EPL	<a href="http://ifamix.sourceforge.net/eclipse/index.html">http://ifamix.sourceforge.net/eclipse/index.html</a>
CAP	GPL	<a href="http://cap.xore.de">http://cap.xore.de</a>
VizzAnalyzer	Evaluation	<a href="http://www.arisa.se/VA.html">http://www.arisa.se/VA.html</a>

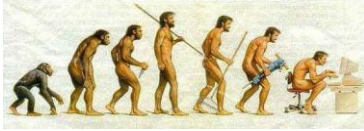
**Table 1 - Tools**

Now, we have to choose the tools to measure the selected metrics. Our set of metric is the following:

- Complexity: MCC, HD, RFC.
- Coupling: ECC, ACC, CBO.
- Cohesion: LCOM, LCOM5, COH.
- Change Impact: WMV.
- Test Coverage: COV.
- Code Quality: CR, CON.
- Encapsulation: DMS.
- Reuse: DIT, NOC.

Seeking metrics in Java tools, the result is the following:

Feature	Metric	Author	Tool	Scope
Complexity	MCC	McCabe	Metrics	Package
	HD	Halstead	Lachesis	Any
	RFC	Chidamber & Kemerer	Lachesis	Class
Coupling	ECC	Martin	Metrics	Package
	ACC	Martin	Metrics	Package
	CBO	Chidamber & Kemerer	Metrics	Class
Cohesion	LCOM	Chidamber & Kemerer	Metrics	Class
	LCOM5	Henderson-Sellers	Eclipse Metrics	Class
	COH	Briand, Daly, and Wüst	Lachesis	Class
Change Impact	WMV	Chidamber & Kemerer	Metrics	Class
Test Coverage	COV	-	Cobertura	Package



Feature	Metric	Author	Tool	Scope
Code Quality	CR	-	CodePro AnalytiX	Any
	CON	-	Checkstyle	Package
Encapsulation	DMS	Martin	Metrics	Package
Reuse	DIT	Chidamber & Kemerer	Metrics	Class
	NOC	Chidamber & Kemerer	Metrics	Class

**Table 2 - Metrics vs Tools**

### 3.1.3.3 Thresholds

Although some accepted standard measures exist, you need to decide what works for you and the acceptable thresholds. Due to that, for every single metric it is mandatory to fix the operational range. We have to find the threshold for every metric. It is summarized as follows:

Feature	Metric	Range	Ideal Value	Operational Range	Threshold
Complexity	MCC	1..∞	1	1..25	25
	HD	0..∞	0	0..38	38
	RFC	0..∞	0	0..72	72
Coupling	ECC	0..∞	1	1..10	10
	ACC	0..∞	1	1..15	15
	CBO	0..∞	1	1..14	4
Cohesion	LCOM	0..∞	0	0..1	1
	LCOM5	0..100%	0%	0..40%	40%
	COH	0..100%	0%	0..40%	40%
Change Impact	WMV	0..∞	0..40	0..40	40
Test Coverage	COV	0..100%	100%	50..100%	50%
Code Quality	CR	0..100%	20%	20..50%	50%
	CON	0..∞	0	0..500	500
Encapsulation	DMS	0..100%	100%	50..100%	50%
Reuse	DIT	0..∞	0	0..7	7
	NOC	0..∞	0	0..10	10

**Table 3 - Metrics vs Thresholds**

### 3.1.3.4 Measurement

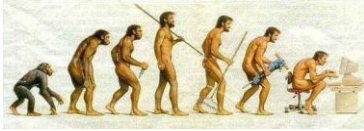
The scope for every metric will be the class. We take all the selected metrics in every class.

### 3.1.3.5 Visualization

Visualization plays a major role in the use of computers to support human reasoning. In this point of the study, we need a way to show our results.

Metrics differ not only in what they measure, but also in the scales against which measurement is performed. As we see in Table 3, there are different scales and ranges for every single metric.

For that reason, we have chosen the radial diagram approach for visualizing maintainability through selected metrics. Radial diagram is used to represent the



relationships of elements (metrics) to a core element (maintainability). This kind of diagram is often named *Kiviati Diagrams*.

Applying radial diagram to this study, an axis will represent the range of values of a single metric. The centre of every axis will mean the worst value for that metric, and the radius will be the ideal value for this metric. We need the thresholds in Table 3 to this representation. The area of the diagram will be the maintainability: the larger is the area, the better maintainability has the system.

Normalization of input data for this kind of diagram is mandatory. We have selected a set of metrics and every one has a scale. We have selected an operational range too (optimal value and threshold). With these data, we can normalize the measurement in the range 0 to 100, where 0 means the worst value and 100 means the best. The formulas for normalization are the following:

If  $V_{th} > V_i$ :

$$N(m) = 100 \cdot \left( 1 + \frac{V_i - m}{V_{th}} \right)$$

... where:

$N(m)$  = normalizes metric between 0 and 100  
 $V_i$  = ideal value for metric  $m$   
 $V_{th}$  = Threshold for metric  $m$   
 $m$  = metric

And if  $V_{th} < V_i$ :

$$N(m) = 100 \cdot \left( \frac{m - V_{th}}{V_i} \right)$$

Consider the following diagrams as examples. The first diagram is a system with very good maintainability in coupling, cohesion and complexity, but it fails in the other quality attributes. In this second diagram, all of the metrics are average. That means that the system under study is quite well implemented because all the internal quality parameters are in the operational range.

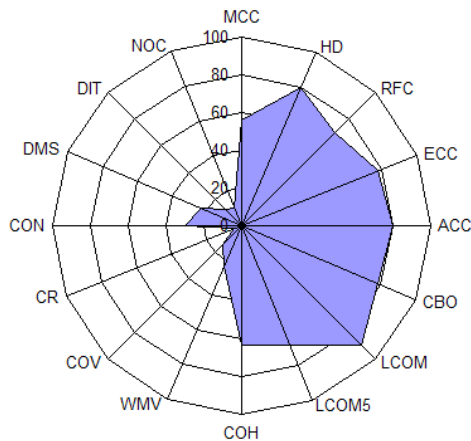
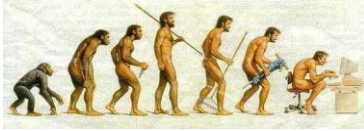


Figure 3 - Radial Diagram Example #1

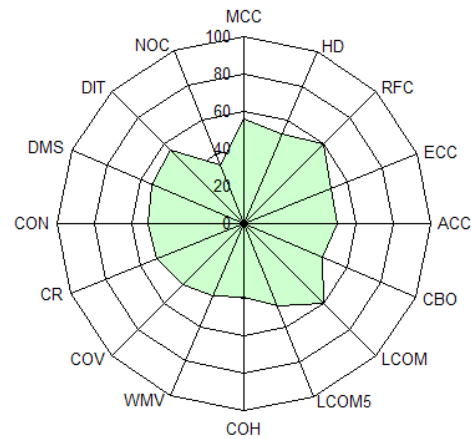


Figure 4 - Radial Diagram Example #2

### 3.1.3.6 Validation

Finally, we have a visual result for maintainability: the area of a radial diagram. To validate this result, we are going to perform one more measurement on our system. This metric is the Maintainability Index [WO95].

→ Maintainability Index (MI) [WO95] is a polynomial metric developed at the University of Idaho, using Halstead's module and McCabe's cyclomatic complexity, plus some other factors. It is used to determine if code has a high, medium or low degree of difficulty to maintain. It ranges from 0 to 100 with larger numbers being better:

- MI < 65: poor maintainability
- 65 ≤ MI < 85: fair maintainability
- MI ≥ 85: excellent maintainability

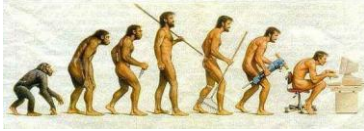
$$MI = 171 - 5.2 \cdot \ln(\text{ave}V) - 0.23 \cdot \ln(\text{ave}V(g')) - 16.2 \cdot \ln(\text{ave}LOC) + 50 \cdot \sin(\sqrt{2.4 \cdot \text{per}CM})$$

... where:

- MI = Maintainability Index
- aveV = average Halstead Volume V per module
- aveV(g') = average extended cyclomatic complexity per module
- aveLOC = the average count of lines of code (LOC) per module
- perCM = average percent of lines of comments per module

### 3.1.4 Case Study

In this case study, an existing product has been evolved to accomplish several goals. The analysed product is a Java-based medical imaging system, of small size (about 10k lines of code). The product is being currently used in several Spanish hospitals. It allows users to visualize multiple medical images at the same time and apply transformations to them. The main objectives to be achieved with this case study are:



- *Improving the usability and performance of the product.* Most complaints from the end users come from the user interface. The overall look & feel of the application should be improved and it should allow some degree of customization. Application responsiveness is another important factor affecting the user's experience, and can be addressed by improving system performance.
- *Improve the maintainability and interoperability of the system.* The original design of the product did not take into account the future needs of the product. It should be refactored to ease future evolution steps and ease its integration with other healthcare systems.

The first step in this case study was the architecture recovery. The recovery process executed was based on QAR (*QUE-es Architecture Recovery*) [Arc06]. QAR is a generic architecture recovery workflow that follows the extract-abstract-present paradigm, and divides the extraction process in three activities (documentation analysis, static analysis and dynamic analysis). QAR offers a process framework for architecture recovery that can be tailored to the specifics of the application.

After that we are looking for a new architecture that supports an increased degree of maintainability and interoperability for the system. These are general good design principles, which are explicitly supported by Service-Oriented Architectures [Erl05]. In the Java context, the OSGi [OSG05] specification provides a functional framework based on these principles.

When we finished the refactoring to SOA, we had two different versions of the same system: one pure Java and other OSGi-based (SOA). We are going to compare the maintainability expected in both systems by applying our prediction model.

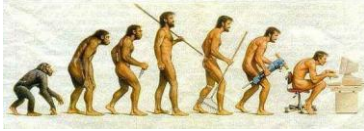
In next tables we summarize the metrics results (mean values). The first table represents the original application. After that, we have collected the metrics in the application, that is, SOA-based application.

Metrics	Value
VG	10
HD	12
RFC	22,28
EC	3,944
AC	10,389
CBO	2,3
LCOM	0,419
LCOM5	59,2
COH	49,01
WMV	16,486
COV	0
CR	20,4
CON	8961
DMS	0,416
DIT	2,76
NOC	0,212

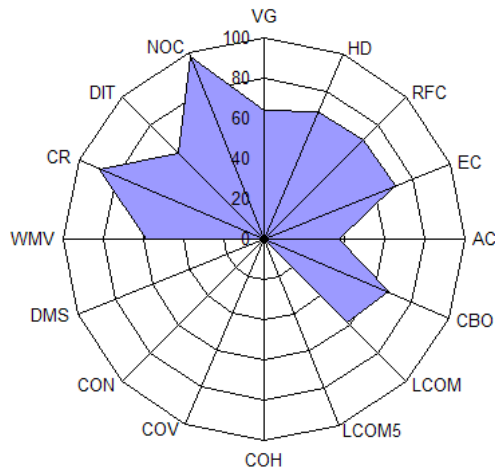
**Table 4 - Metrics in pure-Java application**

Metrics	Value
VG	1,563
HD	8,97
RFC	19,48
EC	2,917
AC	4,875
CBO	1,5
LCOM	0,345
LCOM5	65,98
COH	42,31
WMV	13,75
COV	0
CR	15,5
CON	0
DMS	0,573
DIT	2,029
NOC	0,212

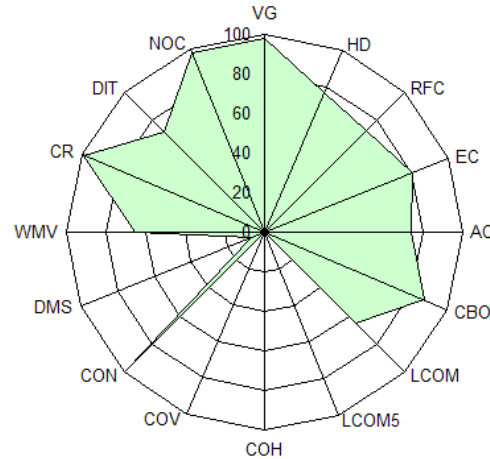
**Table 5 - Metrics in SOA-Java application**



At this point, we can normalize these results and this way we can obtain the radial diagrams:



**Figure 5 - Maintainability in pure-Java application**



**Figure 6 - Maintainability in SOA-Java application**

Comparing both diagrams we can make some conclusions. The first is the SOA-Java application is better (in terms of maintainability) than the original one. The area that represents maintainability is bigger in the SOA-Java.

The new system has improved in almost all quality features: complexity, coupling, change impact, test coverage, code quality, encapsulation and reuse. There are not any changes in test coverage since there is zero test coverage in both applications (pure-Java nor SOA-Java). The cohesion continues being poor in new system. This point should be review in a feature version of the application.

To validate these results, we use the MI metric. This metrics have the following values:

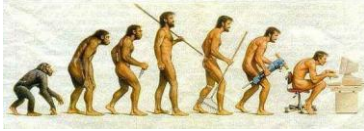
$$\begin{aligned} MI(\text{pure-Java}) &= 82,728 \\ MI(\text{SOA-Java}) &= 89,385 \end{aligned}$$

According to thresholds, the first application is the range of fair maintainability ( $65 \leq MI < 85$ ). SOA-Java application have improved its maintainability, and now is the range of excellent maintainability ( $MI \geq 85$ ).

### **3.2 Test planning - detecting problematic architecture sections**

As software systems grow larger and more complex, their design and implementation becomes more difficult. The use of simple, well defined, and predictable structures such as design patterns can help develop and maintain a software system architecture that fulfils the requirements set to it. Traditionally, software measurement has been applied to code rather than design. However, quality should be incorporated into software as it is being built.

Even more challenges emerge with the maintenance of these systems. Proposed solutions to these problems include refining the software process itself, and increasing



reusability for example through frameworks, design patterns, and product families. Many of these solutions, however, create new problems, most notably by increasing code scattering that may be difficult to overcome. Aspect-oriented programming (AOP) is a technique developed to deal with those design situations, where an ordinary object-oriented solution would result in tangled code. Aspects can encapsulate features that cannot be easily encapsulated by objects or modules.

Design patterns and other complex constructs also make test planning more difficult. We must make sure that additional features use the framework appropriately, and that they do not cause any unwanted side effects (such as breaking any existing pattern). It is vital to pinpoint the impact (in terms of classes or methods) of the modification.

### 3.2.1 Software maintenance with Maisa

Maisa (Metrics for Analysis and Improvement of Software Architectures) is a tool for architecture-based measurement and evaluation of software quality [GPN+02]. The core idea in Maisa is to provide means to analyze the quality of a software system already in its design phase, by computing metrics from its architecture, given as a set of UML diagrams. In addition, the Java-based Maisa tool detects instances of software (design) patterns and anti-patterns. Results can be obtained either visually as diagrams or textually.

To support long-term quality evaluation, Maisa collects historic data and offers a simple facility for statistical analysis. Multiple measurements over the same architecture can be archived, and later used to see how the system has evolved over a given time period. Different architectures can be compared to one another. The statistical facility offers a number of common statistics such as variance.

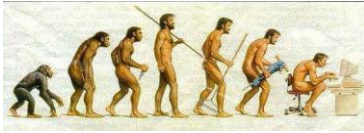
When planning tests with Maisa, the focus should be on the pattern collaboration with respect to the required quality attribute. In a performance critical system, for example, a constant, predictable throughput might be more desirable than a usually large, but variable one. In such a system some patterns like *Mediator* could easily create bottlenecks. Likewise, a predictable, well-known way of creating (*Factory Method* pattern) or accessing (*Visitor* pattern) objects is valuable in itself.

### 3.2.2 Using multiple diagram types

Although class diagrams are the most common ones used in object-oriented designs, any type of diagram or a combination of diagrams can be used to define a design pattern. This is very useful, as different diagrams focus on different aspects of the system. For example, sequence diagrams and state machines model behaviour while class diagrams model structure. In many cases we can achieve more precision by eliminating pattern candidates using multiple diagram types.

Many pattern structures are too general (and thus too numerous) to indicate the presence of an actual instance by themselves. The result is either that we have a large number of candidates only fractions of whom are real instances or we don't have any candidates at all.

We can reduce the number of false positives by verifying potential instances using another diagram type. Consider the class diagram in Figure 7. It shows the structure for an instance of an *Observer* pattern [GHJ+95] while the sequence diagram in Figure 8 shows one possible sequence of events. The structure depicting two inheritance hierarchies having interrelationships is very close to that of several other patterns like



the *Factory Method*. The *Factory Method* implies the creation of objects, which may or may not be depicted on the class diagram. The *Observer*, however, implies only regular method calls.

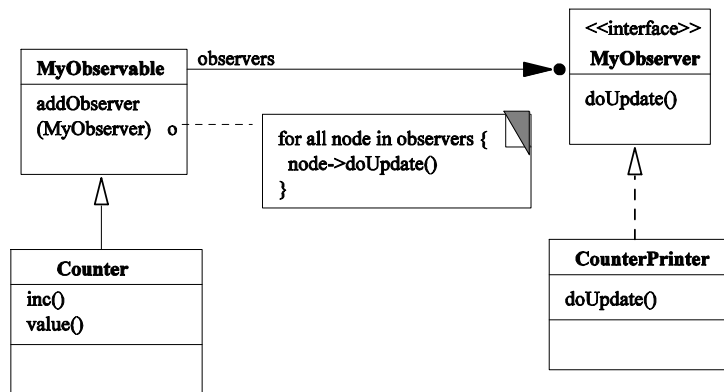


Figure 7 - An instance of an Observer pattern

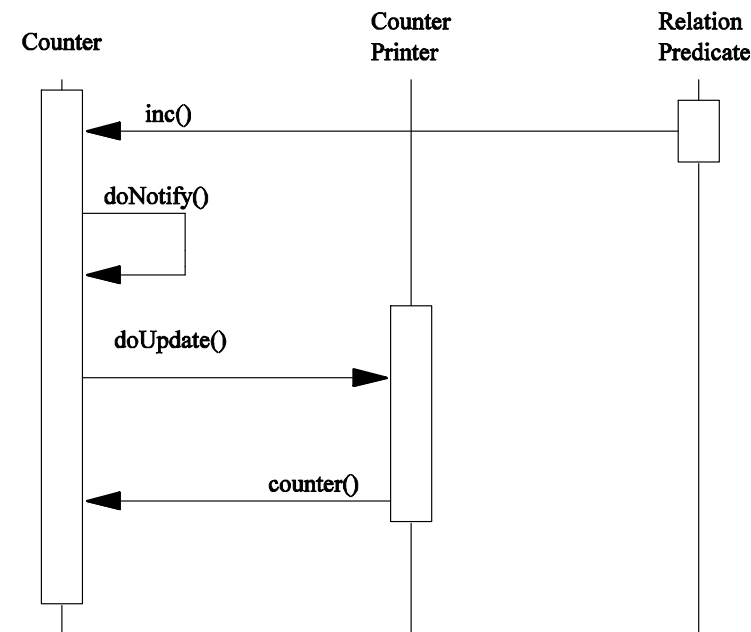
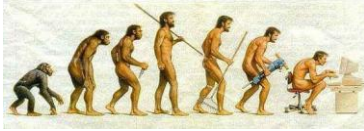


Figure 8 - A sequence diagram for Observer pattern

If multiple diagrams are easily available, they should be used. One way to get dynamic diagrams is to transform use cases or scenarios into collaboration or sequence diagrams.

Our process can be summarized as follows:

- Search those (meta) patterns that are prerequisites for the requested patterns. Dynamically add facts for found instances. This can be done automatically, or the user can choose to act as an oracle stating whether a possible candidate is an actual instance.
- Use these results to prune the search space.
- Confirm the results using available dynamic diagrams.



Less strict pattern definitions may be used in step 2 to increase the number of candidates for step 3.

Dynamic diagrams are usually small compared to class diagrams, so the verification will not be a costly operation. This is especially useful because of the two-phase detection, where the verification only needs to be done for the potential instances acquired as a result from the first detection phase.

### 3.2.3 Case study

In this case study, we had a twofold objective. First, as we weren't familiar with the framework, we wanted to discover the mechanism of how the framework was supposed to be used (i.e. how to add new features). Second, we wanted to ascertain which parts (if any) of the original system were affected by the addition and design test cases accordingly.

We used a graphics framework for the case study. The system provided the base functionality for various types of drawings and diagrams. The variation points offered a way to add new diagrams and other smaller applications to the existing framework. A simple calculator application was added to the framework.

Figure 9 shows the relevant section of the combined system. Those classes that are part of pattern instances across the application-framework interface are shown in black. Note that the term 'pattern' is used in a broad sense here - we can define for example a data access as a pattern. The four classes in the upper right-hand corner belong to the framework, while the rest belongs to the calculator application. We should pay particular attention to the highlighted sections when (re-) testing to ensure the correct functionality over the interface.

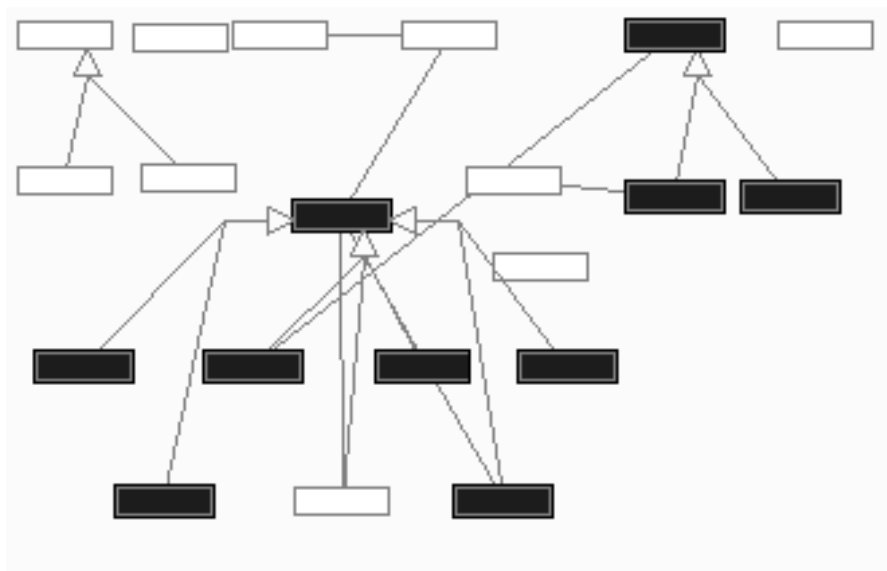
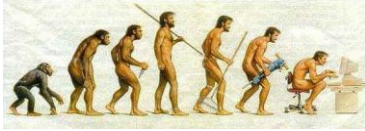


Figure 9 - Affected components after new features were added



## 3.3 Characterizing maintainability across multiple versions

### 3.3.1 Problem at hand

Internal software quality evaluations are quite often subjective and are focused merely on the current state of the system (snapshot evaluations). In contrast, we argue that such evaluations should be objective and repeatable, therefore requiring automated tool support. Additionally, quality evaluations should enable the comparison of the current quality with other software systems or other parts of the same system and previous versions.

- When comparing quality among systems or system parts, one can customize the evaluation (e.g., set acceptable boundaries) according to a larger population of software modules developed using similar processes. In practice, this enables the prioritization of software modules according to the need of quality investments.
- Alternatively, when comparing quality across multiple versions, one can express the current quality relative to previous versions. Such analyses enable to verify whether quality is improving or decreasing, and in result, whether additional investments in quality should be made or not.

In this case study, we have investigated how visualizations can support the evaluation of quality across multiple versions. We refer to this work as *quality trend analysis*. Part of this work can be reused for comparing the quality of multiple systems or multiple system parts.

### 3.3.2 Quality trend analysis

At large, there is a need for analyzing trends in quality at different levels of granularity:

- **Software module level** – Evaluation of the whole population of building blocks with regard to a quality indicator. In Object-Oriented systems, these building blocks can be classes. As an evaluation at this level, we choose descriptive statistics (median, quartile boundaries), as typical when presenting a summary.
- **Building block level** – Evaluation of a single building block, e.g., a class.

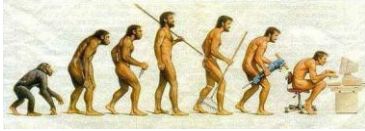
By applying trend analysis at both levels, one can respectively (i) observe trends; and (ii) explain them in terms of specific changes.

In the remainder of this document, we will zoom in on the application of quality trend analysis for a specific set of metrics.

### 3.3.3 Selected metrics

Maintainability is decomposed by [ISO9126] in the following characteristics:

- *Analysability* – the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
- *Changeability* – the capability of the software product to enable a specified modification to be implemented.

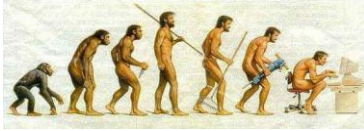


- *Stability* – the capability of the software product to avoid unexpected effects from modifications of the software.
- *Testability* – the capability of the software product to enable modified software to be validated.
- *Maintainability compliance* – the capability of the software product to adhere to standards or conventions relating to maintainability.

Within the scope of this case study, we limit ourselves to *changeability* (also known as *evolvability*) and *testability*.

As metrics for changeability and testability, we select the Chidamber and Kemerer metric suite for Object-Oriented systems [CK94]:

- **Weighted Methods per Class (WMC)** – Measure of the total complexity implemented by the methods in that class. The following forces can help determine the ideal value of this metric:
  - Complexity: The larger the number of methods and their complexity, the more time and effort is required for maintenance.
  - Reuse: The larger the number of methods and complexity, the less likely a class will be reused.
- **Depth of Inheritance Tree (DIT)** – Measure of how many ancestor classes can potentially affect this class. A trade-off can be made among the following forces:
  - Complexity: The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour.
  - Reuse: The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.
- **Number of Children (NOC)** – Measure of how many subclasses are going to inherit the methods of the parent class. Forces:
  - Reuse: The greater the number of children, the greater the reuse.
  - Testability: A class with a large number of children may require more testing of the methods in that class.
- **Coupling between Object Classes (CBO)** – Measure of how many classes are coupled via method or instance variable interaction. Forces:
  - Reuse: The more independent a class is, the easier it is to reuse it in another application.
  - Coupling: The larger the number of couples, the higher the sensitivity to changes in other parts of the design.
  - Testability: The higher the inter-object class coupling, the more rigorous testing needs to be.
- **Response for a Class (RFC)** – Measure of the set of potentially executed methods in response to a message received by an object of that class. Forces:



Testability: If a larger number of methods can be invoked, the testing and debugging becomes more complicated.

Complexity: The larger the number of methods that can be invoked from a class, the greater the complexity.

- **Lack of Cohesion in Methods (LCOM)** – Measure of the absence of similarity among methods with regard to attribute usage. This metric is an inverse cohesion metric: it measures the absence of cohesion. Forces:

Complexity: high lack of cohesion increases complexity.

Table 6 Illustrates the relationship between the selected metrics and maintainability characteristics. These relationships are based on the definitions of changeability and testability.

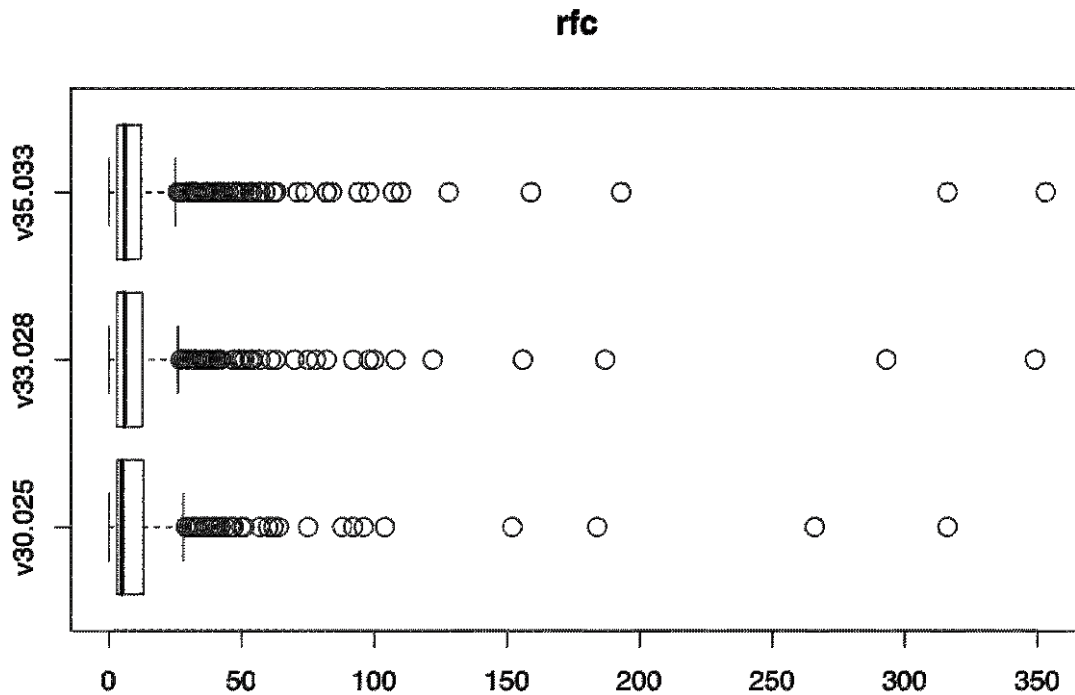
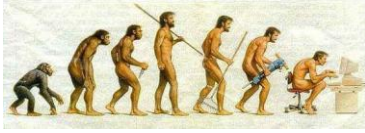
<b>Metric</b>	<b>Subcharacteristic</b>	<b>Characteristic</b>
WMC	complexity reuse	changeability
DIT	complexity reuse	changeability
NOC	reuse	changeability testability
CBO	reuse coupling	changeability changeability testability
RFC	complexity	changeability testability
LCOM	complexity	changeability

**Table 6 - Relation between selected metrics and maintainability characteristics.**

All of these metrics have a negative influence effect on changeability and/or testability. Accordingly, one would wish to minimize their values. With DIT and NOC, however, a low value (e.g., < 7) is desirable.

### 3.3.4 Proposed visualizations

Figure 10 presents our proposed visualization for quality trend analysis at the software module level.



**Figure 10 - Boxplots across three versions.**

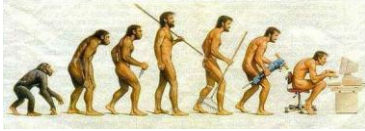
A boxplot is drawn for each version. Such a boxplot is a representation of multiple descriptive statistics:

- **Median:** The bold line in the middle of each box plot represents the median value.
- **Quartile boundaries:** The rectangle surrounding the bold line sets bounds to the first and third quartile. Each quartile represents 25% of the population.
- **Outliers:** Outliers are values that are exceptional (e.g., fall outside the range of +/- 1.5 times the Inter Quartile Range or IQR. IQR is calculated as the difference between the third and first quartile.)

Our proposed visualization for quality trend analysis at the building block level is presented in Figure 11. This visualization consists of two charts: an *upper* and a *lower* chart.

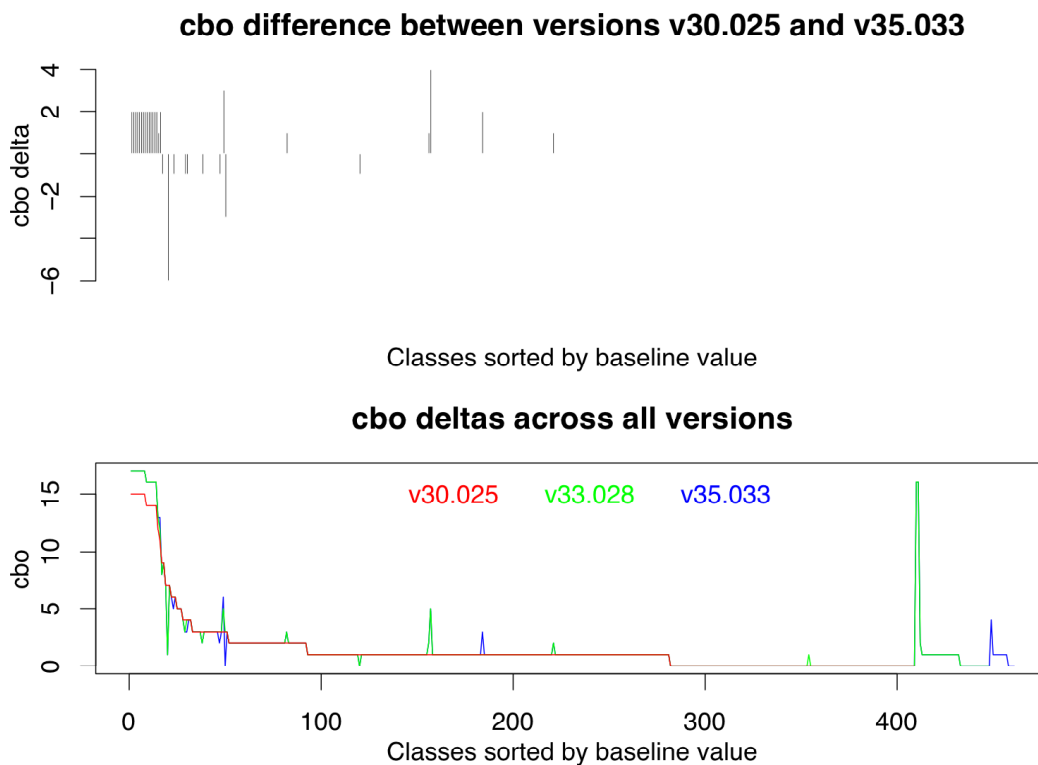
The upper plot is a barplot titled *<Metric name> difference between versions <firstVersion> and <lastVersion>*.

- **X-axis:** each point on the X-axis represents a class. Classes are sorted descending according to the metric value of the class in the first version.
- **Y-axis:** difference between the metric value of a class in the last version compared to its value at the first version (+ => increase, - => decrease)
- **Interpretation:**



Position of the bars: in case most of the bars have a positive value, this indicates that the metric value has increased relatively compared to the first version. Notice that for this global image, you might wish to use the boxplots from the visualization at the software module level.

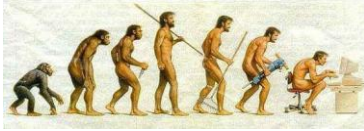
Location of the bars: the location enables you to derive whether the increases/decreases occurred mostly at classes with high/low values at the initial version. In the ideal case, you would wish that the metric values evolved to an average value (not too high, not too low). Visually, this would be represented by negative bars at the left of the X-axis (decrease of high values), and positive bars at the right of the X-axis (increase of low values).



**Figure 11 - Trend analysis bar and lines plot.**

The bottom plot is a "lines plot" titled *<Metric name> deltas across all versions*.

- **X-axis:** Same as the upper plot. The plot of the first version will decrease monotonously across the X-axis, since classes are sorted descending on their metric values of the first version. In case new classes are added in subsequent versions, these are appended to the X-axis in a similar (sorted descending) fashion. Accordingly, for each version, you will find a monotonously decreasing line at the right of the X-axis.
- **Y-axis:** Metric value of a class. Each line represents a single version. The versions are assigned colours from red (first version) to blue (last version). This plot provides a more fine-grained visualization than the upper plot, since it also enables to compare among intermediate versions.
- **Interpretation:**



Trend-analysis: Do the changes to metric values persist across versions, and do they do so in the same direction (e.g., always increasing), or are there shifts among versions (e.g., first increasing, then decreasing)?

Addition of new classes: To the right of the X-axis, you can notice whether a version adds a lot of new classes. E.g., in v33.028 there was an addition of relatively few new classes. From v33.028 to v35.033 there were even less classes added.

### 3.3.5 Application on an industrial case

In October 2007, the authors were asked by Alcatel-Lucent to support monitoring the quality of their modules across their evolution. The main goal was to provide objective data to verify the following assumptions:

- Code quality deteriorates when many people are working in the same code.
- Some modules are in more need of refactoring than others.

Since the developers had an indication that quality was indeed degrading for module Foo (name encoded for confidentiality), we illustrate the application of the visualizations on module Foo.

Three versions were selected: (i) v30.025; (ii) v33.028; (iii) v35.033, corresponding to snapshots from the repository across three different releases (3.0, 3.3 and 3.5). Table 7 presents boxplots for each of the selected metrics (excluding WMC). These visualizations present the following observations:

- **Inheritance**

*DIT*: There is barely any use of inheritance (median DIT of 0). A maximum outlier of inheritance depth of 3 has been introduced between v30.025 and v33.028.

*NOC*: Similar to DIT, we find that inheritance is used rarely (median NOC of 0). Nonetheless, some classes appear to have over 15 subclasses. Between v30.025 and v33.028, changes with regard to inheritance have been performed. From the multi-version boxplot, we might assume that a class with 13 subclasses has been extended with two more subclasses.

- **Coupling**

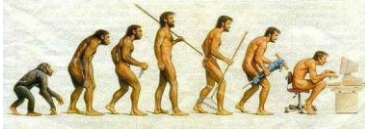
*CBO*: Coupling seems to be quite low (median CBO of 1, outliers starting above 2 coupled classes). Once again, we find that between versions v30.025 and v33.028 some changes have been applied which affect coupling.

- **Complexity**

*RFC*: Complexity has increased slightly after version v30.025 (median increased from 5 to 6). However, the third quartile has reduced slightly across the three versions, possibly indicating that some effort has been spend on reducing complexity.

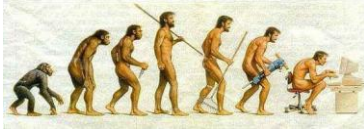
- **Cohesion**

*LCOM*: The median cohesion value has increased, and more extreme outliers have been introduced.



Metric	Including outliers	Excluding outliers
DIT	<p style="text-align: center;"><b>dit</b></p>	<p style="text-align: center;"><b>Normal range of dit</b></p>
NOC	<p style="text-align: center;"><b>noc</b></p>	<p style="text-align: center;"><b>Normal range of noc</b></p>
CBO	<p style="text-align: center;"><b>cbo</b></p>	<p style="text-align: center;"><b>Normal range of cbo</b></p>
RFC	<p style="text-align: center;"><b>rfc</b></p>	<p style="text-align: center;"><b>Normal range of rfc</b></p>
LCOM	<p style="text-align: center;"><b>lcom</b></p>	<p style="text-align: center;"><b>Normal range of lcom</b></p>

**Table 7 - Trend analysis of module Foo at the software module level.**



The increase in all of the metrics seems to indicate that changeability and testability (and therefore maintainability) has reduced somewhat between the three versions. The largest increase can be noted between v30.025 and v33.028.

However, we cannot identify the causes of this maintainability reduction from these boxplots, since they focus on an aggregate level (the software module level). Accordingly, we wish to verify the causes of the observed reduction in maintainability. Therefore, we zoom in on the class level.

- **Inheritance**

- *DIT*: The upper plot indicates that no existing classes have been changed with regard to their inheritance depth. The lower plot indicates that in v33.028, a class has been introduced at an inheritance depth of 3. Accordingly, the outlier introduction observed in Table 7 is due to the introduction of a new class.
- *NOC*: Once again, the upper plot supports us in identifying the cause of the increased inheritance usage. This plot confirms our earlier assumption (made at the software module level) that indeed, a class with 13 subclasses has been extended with two more subclasses. Additionally, two classes, which previously had no subclasses, have been extended in v33.028.
- *Summary*: Existing classes have been reused throughout the observed evolution. However, some quite complex classes have been introduced, since they somehow have to deal with the functionality inherited by no less than 13 ancestors. This seems to be too complex to deal with in an efficient manner, and therefore forms an opportunity for simplification.

- **Coupling**

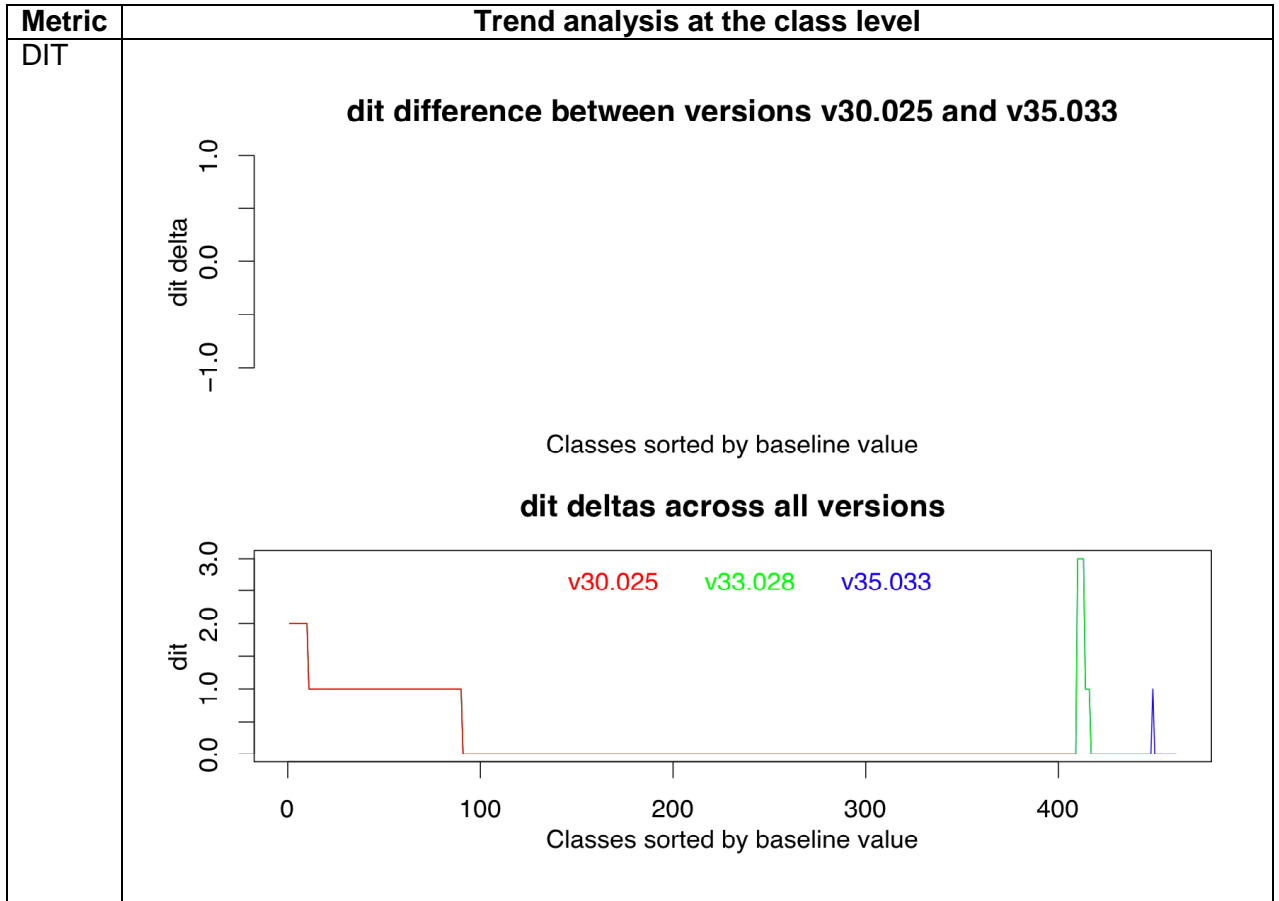
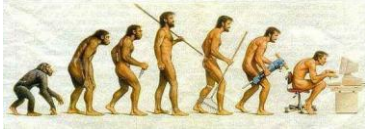
- *CBO*: Quite a number of classes at the high end of the CBO range ( $CBO > 5$ ) have been extended with even more dependencies on other classes. For most of these extensions, this was due to an increase in v33.028. Occasionally, however, this increase in coupling has continued in v35.033. We would advise to study this instability more closely to verify the need for decoupling, e.g., by splitting up a class according to its various dependencies.

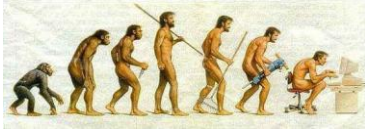
Moreover, in v33.028, a class was introduced having dependencies on no less than 15 other classes. This seems to indicate the implementation of a *quick hack*.


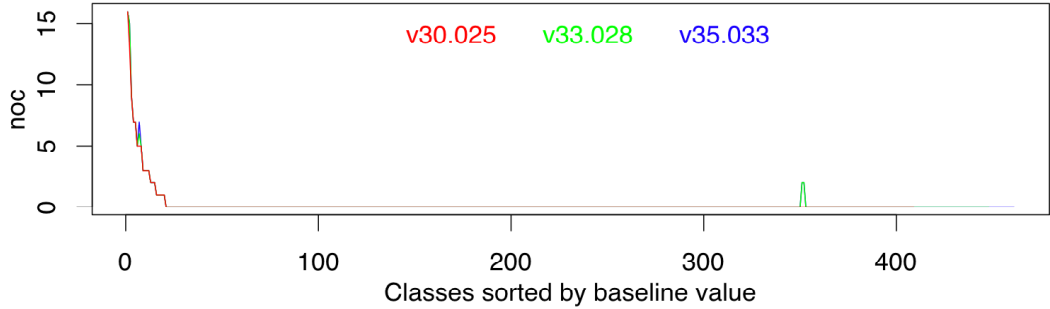
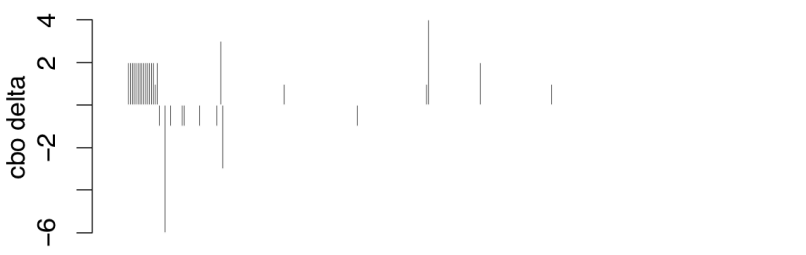
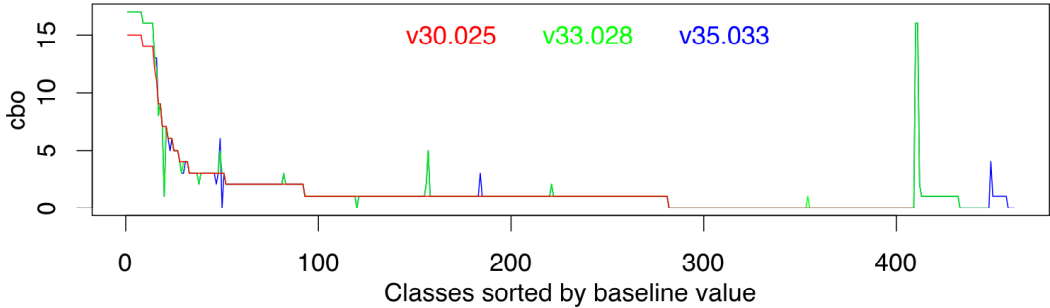
- *Summary*: We confirm the initial observation that coupling has been increased, especially for classes who were already had quite a number of dependencies.

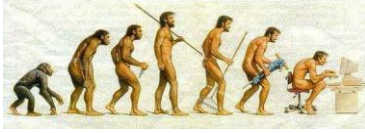
- **Complexity**


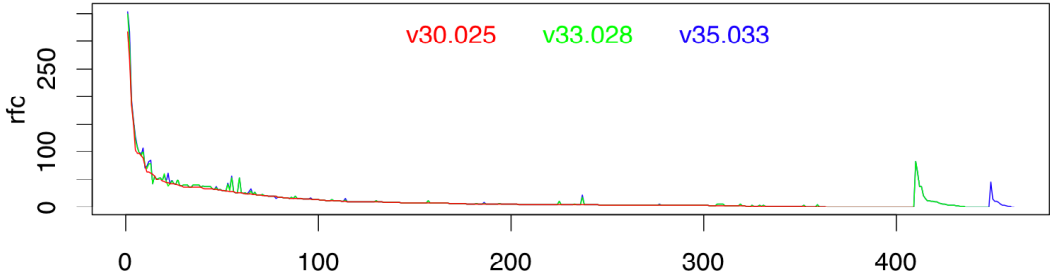
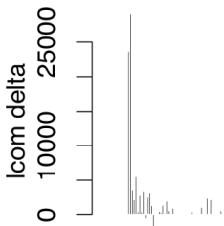

- *RFC*: Once more, the upper plot indicates increases (of up to 50 additional invoked methods) mostly in those classes who already had quite high values. Moreover, these increases tend to continue across the versions.
- *Summary*: Complexity tends to increase across the three versions. Complex classes seem more likely to become even more complex in future releases.

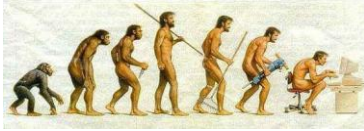




Metric	Trend analysis at the class level
NOC	<p data-bbox="539 353 1337 387"><b>noc difference between versions v30.025 and v35.033</b></p>  <p data-bbox="751 689 1126 723">Classes sorted by baseline value</p> <p data-bbox="715 752 1163 786"><b>noc deltas across all versions</b></p>  <p data-bbox="751 1081 1126 1115">Classes sorted by baseline value</p>
CBO	<p data-bbox="539 1216 1337 1249"><b>cbo difference between versions v30.025 and v35.033</b></p>  <p data-bbox="751 1552 1126 1585">Classes sorted by baseline value</p> <p data-bbox="715 1615 1163 1648"><b>cbo deltas across all versions</b></p>  <p data-bbox="751 1944 1126 1977">Classes sorted by baseline value</p>



Metric	Trend analysis at the class level
RFC	<p data-bbox="549 353 1329 387"><b>rfc difference between versions v30.025 and v35.033</b></p>  <p data-bbox="751 689 1126 723">Classes sorted by baseline value</p> <p data-bbox="724 748 1153 781"><b>rfc deltas across all versions</b></p>  <p data-bbox="751 1084 1126 1117">Classes sorted by baseline value</p>
LCOM	<p data-bbox="533 1216 1345 1249"><b>lcom difference between versions v30.025 and v35.033</b></p>  <p data-bbox="751 1552 1126 1585">Classes sorted by baseline value</p> <p data-bbox="708 1610 1169 1644"><b>lcom deltas across all versions</b></p>  <p data-bbox="751 1946 1126 1980">Classes sorted by baseline value</p>



- **Cohesion**

- *LCOM*: The upper plot for LCOM provides the clearest example of the pattern we observed for the previous metrics: classes with high values for a metric tend to have their value increased in next releases.

### 3.3.6 Interpretation of results

When presenting these results to the developers, they generally found the charts useful. Nonetheless, the following remarks were made:

- *“The interpretation of these plots requires training.”*

This is certainly the case, and this case study report provides an initial formalization of essential documentation.
- *“To identify the causes of reduced maintainability, one needs interactive charts that can provide the name of a class when hovering over it with the mouse.”*

Initially, we envisioned complementing these visualizations with summaries of the changes among the versions providing the class names. However, we agree that interactive charts present additional value.
- *“For evaluations across many versions, it might be useful to use time as the third dimension.”*

In case of many versions (e.g., > 10), the following extensions can be made:

*Time as the third dimension* – This has the disadvantage that in case values increase and later on decrease, “mountain tops” might appear on the chart, hiding observations from later versions. Then again, since our initial observations tend to indicate that increasing values lead to even more increases, monotonous increases seem more likely.

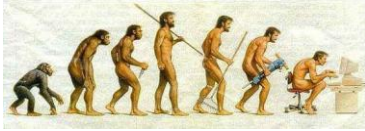
*Sliding through a window of versions* – This has the disadvantage that the developer needs to remember the original values.

Currently, we delay the extension of these visualizations until observations have been made on other software modules.

### 3.3.7 Conclusion

The main contribution of this work is a pair of visualizations to support quality trend analysis. Accordingly, this work paves the way for extensive empirical analysis on quality trending. Indeed, our limited set of observations already suggests a pattern. It seems that a software system can be divided in a stable and an instable part. Classes in the instable part tend to continue to change across the evolution of the system.

This case study illustrated that, indeed, quality trend analysis can be partially automated. This automation enables to evaluate quality in an objective, repeatable manner.



## 4. Predicting maintainability

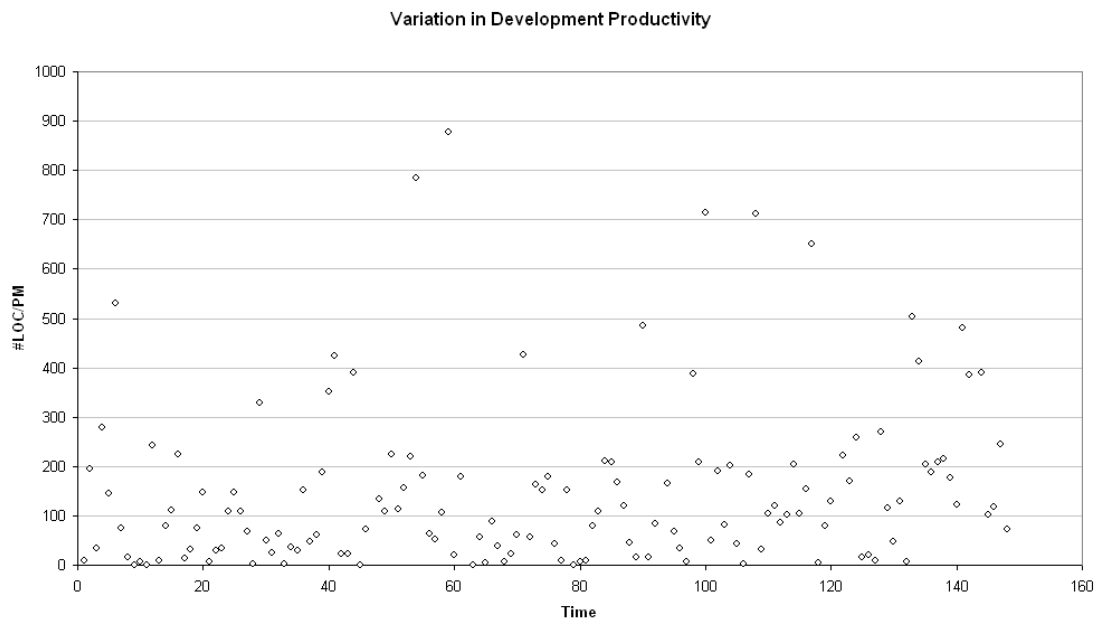
The following two case studies elaborate on the true goal of proactive quality management: to be able to predict the end result, and take action accordingly.

### 4.1 Explaining development productivity variation using early metrics

#### 4.1.1 Problem at hand

To estimate the time required for developing a new feature, most estimation models rely on historic data with regard to development productivity. Typically, development productivity is expressed in terms of numbers of KLOC per person-year [Sne97], and it is mostly weighted using a plethora of factors. However, the internal structure of the software system part to be extended is hardly ever taken into account. In other words, typical development effort estimation models consider development productivity as a stable factor.

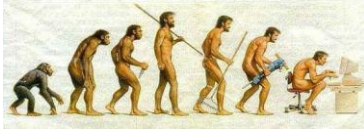
Quite in contrast, we noted considerable variability concerning the productivity across a history of 148 features developed at Alcatel-Lucent, as illustrated in Figure 12. This variability is explained for merely 40.4% by the number of lines of code affected by the feature.



**Figure 12 - Development productivity variation across time.**

Accordingly, it is reasonable to assume that other factors exist that can explain the 59.6% of variability not explained by the number of lines of code. In other words, there is considerable room for improving estimation accuracy.

Sadly, new candidate estimation factors should be limited to metrics that can be calculated at estimation time. While internal quality of the affected modules seems to



be a major explaining factor, data bookkeeping is more than often not so detailed as to trace feature data to a certain source code scope. Indeed, in our case, the recorded data is limited to: (i) the number of lines of code affected; and (ii) the number of modules affected.

Accordingly, this case study sets out to characterize the extent to which the *number of modules affected by a feature* can lever the accuracy of traditional effort estimation models that rely solely on the number of affected lines of code.

The solution to this problem requires creating and evaluating statistical models for the available historic data. An improvement is achieved in case of an increase of the amount of variation in development productivity explained from the list of factors used.

#### 4.1.2 The baseline statistical model

The baseline statistical model used traditionally to explain the variability in development effort consists merely of one factor, namely the number of lines of code affected by the feature. Indeed, the number of lines of code is recognized as one of the major cost factors, as amongst others illustrated in COCOMO [MM85]. The most primitive model is depicted in Table 8.

Development effort (in PM) =  $locIndependentEffort + locFactorSize * LOC + error$

Where

- *locIndependentEffort* is the effort that is present even in case zero lines of code would be affected. This incorporates overhead that is always present, e.g. for communication.
- *locFactorSize* represents the increase in person-months in case one additional line of code is affected.
- *error* represents the variability that is not explained by the above factors.

**Table 8 - Effort explained by number of lines of code only.**

An evaluation of this baseline model on the historic data is presented in Table 9. This evaluation indicates that the number of lines of code is a statistically significant predictor of the development effort of a feature. In other words, it is very unlikely (chance < 0.01%) that this correlation between the number of lines of code and development effort is purely a coincidence.

```
lm(formula = effort ~ loc)
```

```
Residuals:
```

```
   Min       1Q   Median       3Q      Max
-83.511 -14.602  -7.247   4.915 152.425
```

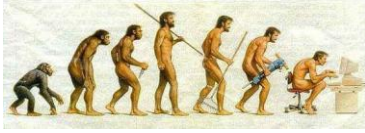
```
Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.741e+01  2.953e+00   5.895 2.33e-08 *** ← locIndependentEffort
loc          3.359e-03  3.284e-04  10.228 < 2e-16 *** ← locFactorSize
```

```
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 31.67 on 152 degrees of freedom
```



Multiple R-Squared: 0.4077, Adjusted R-squared: **0.4038**  
F-statistic: 104.6 on 1 and 152 DF, p-value: < 2.2e-16

**Table 9 - Evaluation of the baseline model.**

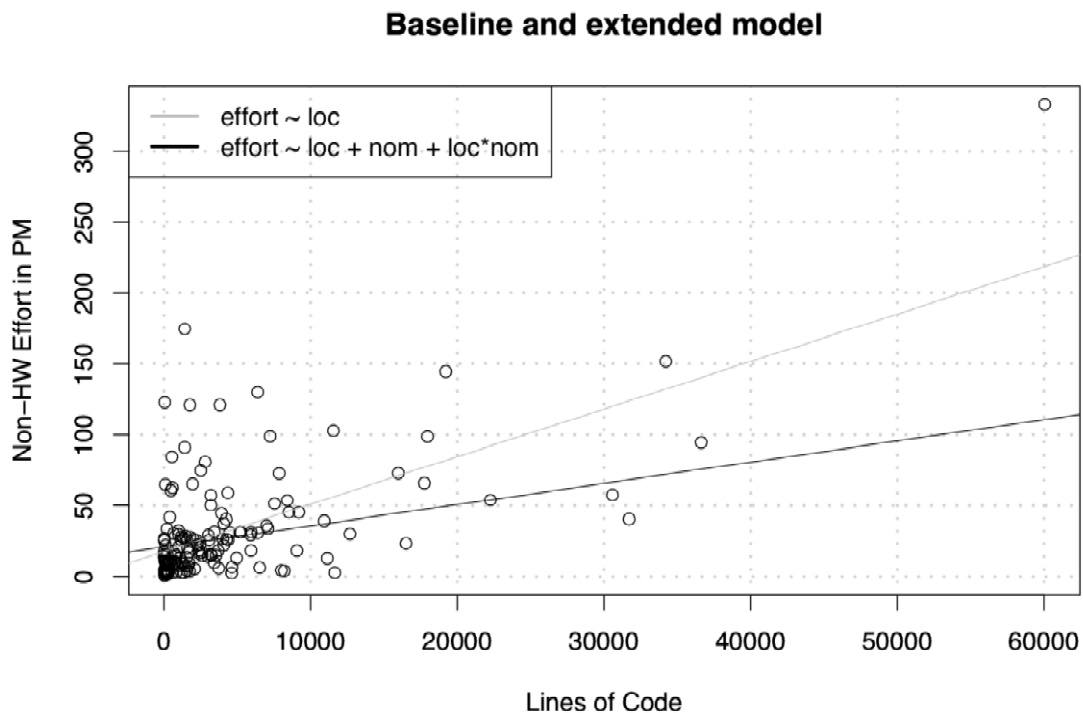
The resulting model is indicated to explain 40.38% of the variation in development effort. The values of the factors locIndependentEffort and locFactorSize present the following formula to estimate development effort based solely on the number of affected lines of code:

$$\text{effort} = 17.41\text{PM} + 0.00359\text{PM} * (\# \text{LOC affected})$$

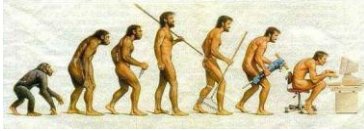
or alternatively:

$$\text{effort} = 17.41\text{PM} + 3.59\text{PM} * (\# \text{KLOC affected})$$

A visualization of this model is presented in Figure 13. Feature data points are indicated as circles with X and Y coordinates. The X coordinate represents the number of lines of code affected by that feature. The Y coordinate represents the actual development effort required by that feature. The grey regression line represents the baseline model. Accordingly, the (vertical) distance between a point and the regression line illustrates the over/underestimation by the model.



**Figure 13 - Scatterplot of the relation between development effort in PM and lines of code as estimated by the baseline model (grey regression line).**



Thus, the linear model indicates that most of the features require at least 17.41PM, and that the development of each Kilo Lines of Code affected by a feature will take 3.6 person-months. Note: this linear model is merely valid for the set of features contained in the data set. There are no guarantees that this linear model provides the same explanatory power for other features.

### 4.1.3 Extending the baseline model

When incorporating *the number of modules affected by a feature* in the baseline model, the model will contain two factors. The effect of these factors should not only be considered in isolation, one must also take into account interactions between these two factors.

An *interaction effect* exists in case the effect of one factor is dependent upon the value of another factor. E.g., the number of affected modules might not influence the effort required to develop features affecting a very high number of lines of code.

Accordingly, we extend our statistical model to three factors, as illustrated in Table 10.

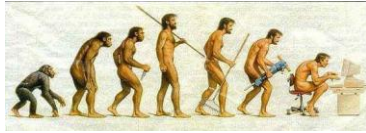
$$\text{Development effort (in PM)} = \text{independentEffort} + \text{locFactorSize} * \text{LOC} + \text{modulesFactorSize} * (\#\text{modules}) + \text{interactionFactorSize} * (\text{LOC} * \#\text{modules}) + \text{error}$$

Where

- *independentEffort* once again equals the effort that is present even in case zero lines of code and zero modules would be affected. This incorporates overhead which is always present, e.g. for communication.
- *locFactorSize* represents the increase in person-months in case one additional line of code is affected (while the number of modules remains unchanged).
- *modulesFactorSize* represents the increase in person-months in case one additional module is affected (while the number of lines of code remains unchanged).
- *interactionEffectSize* represents the increase in person-months in case one additional line of code/module is affected at a specific number of lines of code/modules.
- *error* represents the variability that is not explained by the above factors.

**Table 10 - Effort explained by number of lines of code only.**

An evaluation of this model is presented in Table 11. In relation to the baseline model, *locFactorSize* is reduced from 3.59 to 1.502. Interestingly, the number of modules does not have a significant main effect on development effort, but it does have a significant interaction effect with the number of lines of code.



```
lm(formula = effort ~ locIndependentFactor + locFactorSize
    + modulesFactorSize * interactionEffectSize)

Residuals:
    Min       1Q   Median       3Q      Max
-70.444 -16.713  -7.259   4.453 151.556

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.063e+01  3.759e+00  5.488 1.69e-07 ***
locFactorSize  1.502e-03  6.008e-04  2.499 0.01351 *
moduleFactorSize  2.045e-01  5.250e-01  0.390 0.69743
interactionEffectSize  5.196e-05  1.623e-05  3.201 0.00167 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 30.5 on 150 degrees of freedom
Multiple R-Squared:  0.4578,    Adjusted R-squared:  0.447
F-statistic: 42.22 on 3 and 150 DF,  p-value: < 2.2e-16
```

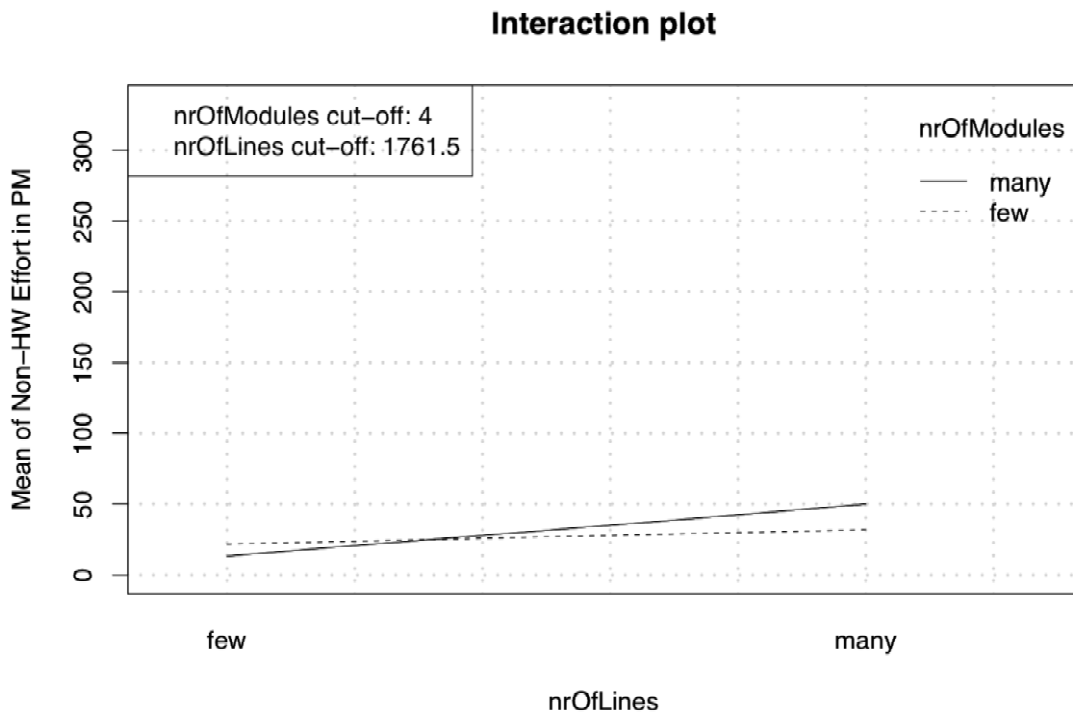
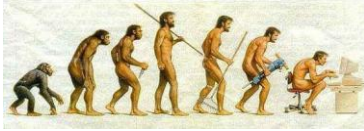
**Table 11 - Evaluation of the model including number of affected modules.**

To characterize the interaction effect, we study the factors *number of lines of code*, and *number of modules* in two levels each:

- few: value is less than or equal to the median
- many: value is greater than the median

Accordingly, we ask ourselves whether development effort differs between features affecting *few* or *many* lines of code, and whether these differences depend on whether these features affect *few* or *many* modules. The answer to this question is best illustrated in an *interaction plot*, as illustrated in Figure 14.

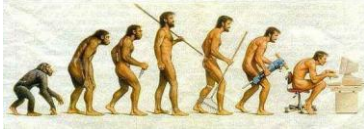
Figure 14 indicates that features affecting many lines of code (> 1762) only require more development effort than features affecting few lines of code in case many modules are affected. Indeed, as depicted in Figure 15, there is limited overlap between the two boxplots of the features affecting many modules (boxplots 3, median/mean development effort = 10/13 PM and boxplot 4, median/mean 34/50 PM). This means that among the features affecting many modules, the number of affected lines of a code has a considerable effect on development effort.



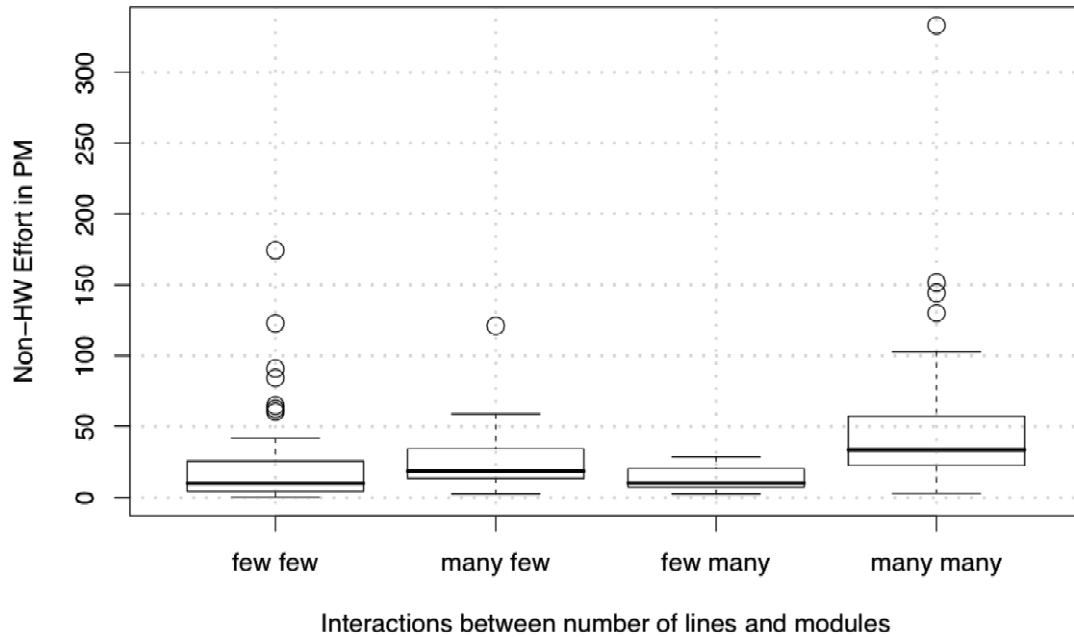
**Figure 14 - Interaction plot between number of lines of code and number of modules affected by a feature.**

In contrast, among the features affecting merely few modules, the number of lines of code barely affects development effort. This can be observed from the considerable overlap between the two boxplots for features affecting few modules (boxplot 1, median/mean 11/22 PM and boxplot 2, median/mean 18/32 PM).

In summary, we can state that *the number of modules affected by a feature* modulates the effect of *the number of affected lines of code* on development effort.



### Effort across factor interactions



**Figure 15 - Boxplot of development effort for the four combinations of the two factors in the extended model.**

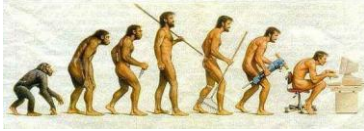
The degree of variation in development effort that can be explained from this extended model has risen with 4% from 40.8 to 44.7%. When filling in the values of the factor estimates, we achieve the following formula to estimate development effort:

$$\begin{aligned} \text{effort} = & 20.63\text{PM} + 0.001502\text{PM} * (\# \text{LOC affected}) \\ & + 0.2045 \text{PM} * (\# \text{modules affected}) \\ & + 0.00005196\text{PM} * (\# \text{LOC affected} * \# \text{modules affected}) \end{aligned}$$

or alternatively:

$$\begin{aligned} \text{effort} = & 20.64\text{PM} + 1.502\text{PM} * (\# \text{KLOC affected}) \\ & + 0.2045 \text{PM} * (\# \text{modules affected}) \\ & + 0.05196\text{PM} * (\# \text{LOC affected} * \# \text{modules affected}) \end{aligned}$$

A visualization of this model is presented in Figure 13. The grey line represents the baseline model and the black line represents the extended model. The figure indicates that a considerable amount of the variation in development effort (approximately 55%) can still not be explained, and that most of this variation concerns features affecting less than 10KLOC.



#### 4.1.4 Interpretation of Results

Yes, the variation in development productivity can be more accurately explained by the combination of (i) the number of affected lines of code; and (ii) the number of affected modules than merely by the number of affected lines of code. More specifically, features affecting more modules relatively require more effort.

However, at 4%, this increased accuracy is quite small. This limited improvement can be explained by the intrinsic relation between the number of affected modules and the number of affected lines of code: most features affecting many modules also affect many lines of code.

To illustrate the extent of the contribution of the number of modules on development effort (and therefore indirectly on development productivity), consider the following example.

A feature is planned, and the experts estimate that it will affect 1KLOC. Depending on the number of modules across which the effect of this feature is distributed, the development effort and productivity are predicted in Table 12.

#modules	Development effort	Development productivity
1	22.39 PM	44.7 LOC/PM
2	22.64 PM	44.2 LOC/PM
3	22.90 PM	43.7 LOC/PM
4	23.16 PM	43.2 LOC/PM
5	23.41 PM	42.7 LOC/PM

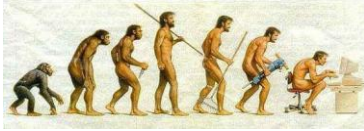
**Table 12 - Effort and productivity in function of the number of modules.**

Accordingly, the effect of the number of modules is quite small. Nonetheless, this effect will be noticeable for large features.

#### 4.1.5 Conclusion

By including the number of modules a feature might affect in the estimation of its development effort, the estimation accuracy will improve slightly (4%). However, this still leaves about 55% of the variation in development effort unaccounted for, urging the need to refine data bookkeeping as to enable the incorporation of internal quality metrics in the estimation models.

Moreover, we have observed that the number of modules affected by a feature modulates the effect of the number of lines of code on development effort. From a pragmatic point of view, this interaction effect can be interpreted as follows. For features affecting merely few lines of code (in our data, less than 1.8KLOC), the development effort does not seem to increase much when the change is not encapsulated in few modules. However, this encapsulation only becomes important for features affecting many lines of code (> 1.8KLOC). Accordingly, for those features affecting many lines of code across many modules, considerably more effort should be scheduled.



## 4.2 Release planning - estimating resource cost of CR and new reqs

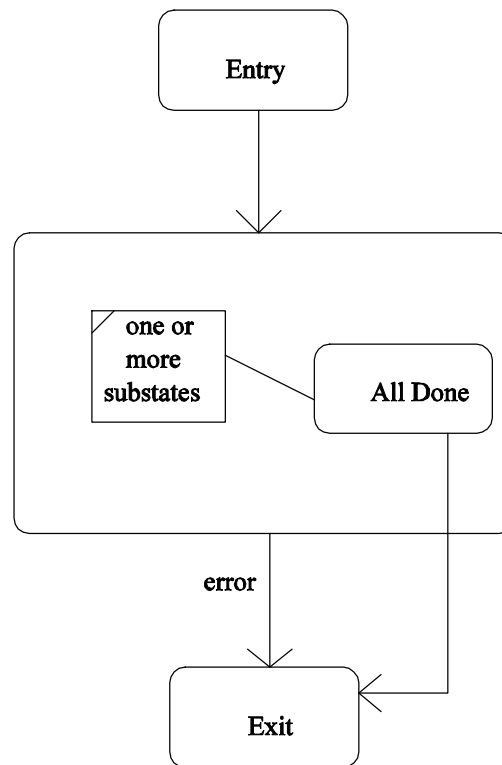
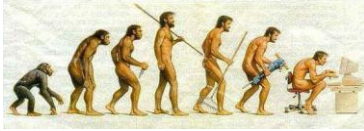
Estimating the cost of future releases is an integral part of the development of software product lines. Architectural design may try to predict coming requirements using techniques such as feature modelling. Moreover, different cost analysis models can be used to estimate the cost of change. These models usually rely on collecting statistics such as interface complexities or function points, and our main goal is to provide this kind of information. We discuss some of our approaches and present a case study.

### 4.2.1 Aspect-oriented features

Although we originally directed our attention towards object-oriented design patterns, some of the early tests strongly indicated both the need to look for aspect-like features in software architectures and the suitability of dynamic diagrams in finding them. Especially the larger scale industrial cases we have studied support this. Many domain-specific patterns have proven to be aspects rather than design patterns. On the other hand, aspect-based solutions need to be designed properly for easy maintainability and testability (cf. section 3.2 page 18).

The larger an organization or a software product is, the more important it is to ensure that frequent, in a sense standard operations are always performed in the same way. For example, a product family might well need a single common way for error handling. However, using design patterns to implement such features is not necessarily a good idea.

Error finding and correcting as well as producing log files are typical examples of features that imply a particular behaviour common to most if not all objects in the system. Figure 16 shows a simplified example of an error-handling mechanism. Note that both outgoing transitions from the middle node pass through the exit node. Also, there should only be one each of the depicted transitions.



**Figure 16 - A possible error-handling mechanism in a state diagram**

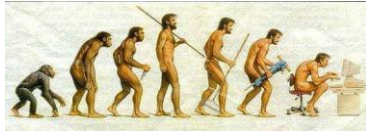
#### 4.2.2 Round-trip analysis

Although Maisa is primarily developed for early quality evaluation, it can also be utilized in later phases of the software life cycle. When an existing system is being redesigned, we can analyze the old system and compare the results against those of the new system. If we are using a large amount of legacy code and are unfamiliar with the design, this can be helpful.

If Maisa is used as a reverse or re-engineering aid, we may want to know the main design elements of the existing software. This way, we can detect hot spots as well as potential bottlenecks in the design. For example, the presence of design patterns can give us a clue as to how the designer intended the program to be modified (with patterns such as *Visitor* or *Strategy*). Conversely, the lack of *Builder* or *Abstract Factory* patterns, for instance, may point out the need for refactoring (the cost of which is usually hard to predict).

Whenever additional information is available (e.g., when we use data that is extracted from program code instead of design data), we may use a different set of constraints for each pattern. Exact data will naturally yield a more precise result. This approach can be utilized e.g. when using Maisa collaboratively with a reverse engineering tool such as Columbus [FGM+02].

However, we need to ensure that all the required information is present. This can be a problem if, for instance, the reverse engineering system has been intended for visualisation purposes. Additionally, any language dependent characteristics, such as C++ macros and template functions, have to be taken into account. These features may not be enough to prevent us from detecting particular patterns, but they may cause inaccuracies in some cases, particularly if they occur frequently.



While we may use complete information whenever possible, using incomplete information (and getting false positives) is not necessarily a bad thing. We may want to see those instances that are almost design patterns but not quite. The user can then decide, whether the system should be modified and take appropriate actions.

### 4.2.3 Case study

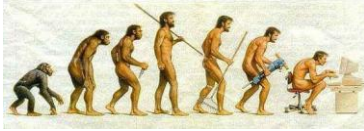
The diagram representation in Maisa is implemented with a framework mechanism (allowing new diagrams to be added easily). In this case study, we wanted to find out whether it would be feasible to try out alternative ways to implement the concept of diagram. The task was to estimate the size of the reuse interface of the framework. For this purpose we looked for instances of meta patterns as the variation points in our framework were implemented with meta patterns - more specifically *Unification* [Pre95].

We used an early version of Maisa as a case study. This version had 325 classes containing some 60000 lines of Java code. A complete scan took a little under two minutes.

Table 13 shows a summary of the detected instances. The number of *Unification* pattern instances corresponds to the number of variation points in the reuse interface, which makes it relatively compact, and several alternatives have since been implemented with moderate effort.

meta pattern	number of instances
Indirect Read	3563
Indirect Write	450
Single Reference	422
Multiple Reference	105
Unification	10
Container	26

Table 13 - Meta pattern instances



## 5. Summary

This document has introduced some case studies concerning an important quality indicator in software intensive systems: the maintainability. This section shows a brief summary of the main contributions of every single study.

### 5.1 Assessing maintainability using metrics

This case study introduces a method to estimate and evaluate the maintainability of a software system using internal metrics.

The first action to perform this case study is making a relation between maintainability features (analysability, changeability, stability and testability) and internal quality indicators (complexity, coupling, cohesion, change impact, test coverage, code quality, encapsulation and reuse). This relation has been shown at Figure 1.

After that, it is proposed the following method to measure maintainability of a Java system:

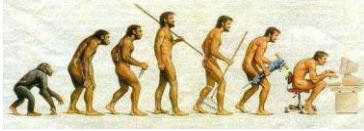
- *Selecting metrics:* complexity (MCC, HD, RFC), coupling (ECC, ACC, CBO), cohesion (LCOM, LCOM5, COH), change impact (WMV), test coverage (COV), code quality (CR, CON), encapsulation (DMS) and reuse (DIT, NOC)
- *Selecting tools:* Metrics, Lachesis, Eclipse Metrics, Cobertura, CodePro AnalytiX and Checkstyle
- *Calculating thresholds:* See Table 3.
- *Measurement:* Taking metrics with selected tools.
- *Visualization:* Using radial diagrams.
- *Validation:* Using MI (Maintainability Index) metric.

### 5.2 Test planning - detecting problematic architecture sections

Maisa (Metrics for Analysis and Improvement of Software Architectures) is a tool for architecture-based measurement and evaluation of software quality. Maisa predicts the quality of a software system in an early design phase, by computing metrics from its architecture, given as a set of UML diagrams.

### 5.3 Characterizing maintainability across multiple versions

This industrial case study presents a method to characterize changeability and testability using Chidamber and Kemerer object-oriented metrics: WMC, DIT, NOC, CBO, RFC and LCOM. For visualization results boxplots (see Figure 10) and lines plots are used. The main contribution of this study is a pair of visualizations to support quality trend analysis.



## 5.4 Explaining development productivity variation using early metrics

Most estimation models rely on historic data with regard to development productivity for calculating Development effort:

Development effort (in PM) =  $\text{locIndependentEffort} + \text{locFactorSize} * \text{LOC} + \text{error}$

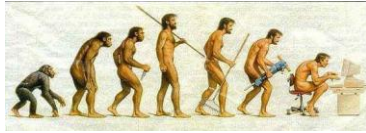
In this case study, the number of modules affected by a feature is incorporated as follows in the baseline model:

Development effort (in PM) =  $\text{independentEffort} + \text{locFactorSize} * \text{LOC} +$   
 $\text{modulesFactorSize} * (\#\text{modules}) +$   
 $\text{interactionFactorSize} * (\text{LOC} * \#\text{modules}) +$   
 $\text{error}$

This study concluded that by including the number of modules a feature might affect in the estimation of its development effort (4% improved accuracy). Another important result is that the number of modules affected by a feature modulates the effect of the number of lines of code on development effort.

## 5.5 Release planning - estimating resource cost of CR and new reqs

This case study presents a way for estimating the cost of future releases in software product lines. One way for obtain better results in terms of maintainability and testability is using aspect-based solutions. The Maisa tool can be used in later phases of the software life cycle for comparing versions and changes in the design in software systems.



## 6. References

[AC94] Brito e Abreu F. and Carapuça R. "Object-Oriented Software Engineering: Measuring and controlling the development process". 4th Int Conference on Software Quality, Mc Lean, Va, USA. 1994.

[Arc06] Arciniegas, J.L, "Contribution to Quality-driven Evolutionary Software Development Process for Service-Oriented Architecture", Ph. D. Thesis, Polytechnic University of Madrid, 2006.

[AW03] M. Alshayeb and L. Wei, "An empirical validation of object-oriented metrics in two different iterative software processes," IEEE Transactions on Software Engineering, vol. 29 (11), 2003.

[BDW98] Briand, L. C., Daly, J. W., and Wüst, J., "A Unified Framework for Cohesion Measurement in Object-Oriented Systems", Empirical Software Engineering, vol. 3, no. 1, 1998, pp. 65-117.

[BK95] Bieman, James M. & Kang, Byung-Kyoo. Cohesion and reuse in an object-oriented system. Proceedings of the 1995 Symposium on Software. Pages: 259 - 262. ISSN:0163-5948. ACM Press New York, 1995.

[CK94] S. Chidamber and C. Kemerer, "A Metrics Suite for Object-Oriented Design," IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493, June 1994.

[CKK+02] M. Ajmal Chaumon, Hind Kabaili, Rudolf K. Keller and François Lustman. "A Change Impact Model for Changeability Assessment in Object-Oriented Software Systems". Département IRO Université de Montreal. Science of Computer Programming, Volume 45, Number 2, November 2002 , pp. 155-174(20)

[EGF+04] Letha H. Etzkorn, Sampson E. Gholston, Julie L. Fortuneb, Cara E. Steina, Dawn Utleby, Phillip A. Farringtonb and Glenn W. Coxa. "A comparison of cohesion metrics for object-oriented systems". Department of Computer Science and Department of Industrial and Systems Engineering and Engineering Management, The University of Alabama, USA. January 2004.

[Erl00] Erlikh, L. "Leveraging legacy system dollars for E-business". IEEE IT Pro, May/June 2000.

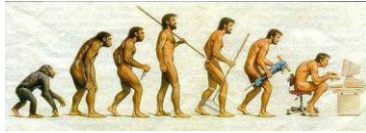
[Erl05] Erl, Thomas. "Service-Oriented Architecture: Concepts, Technology, and Design". Upper Saddle River: Prentice Hall, 2005.

[Fdc07] Free On-line Dictionary Of Computing. <http://foldoc.org/>

[Fen94] N. Fenton, "Software Measurement: A Necessary Scientific Basis," IEEE Transactions on Software Engineering, vol. 20 (3), pp. 199-206, 1994.

[FGM+02] Ferenc, R., Gustafsson, J., Müller, L., and Paakki, J.. Recognizing Design Patterns in C++ Programs with the Integration of Columbus and Maisa. Acta Cybernetica 15(4):669-682, 2002.

[GHJ+95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.



[GPN+02] Gustafsson, J., Paakki, J., Nenonen, L., and Verkamo, A.I. Architecture-Centric Software Evolution by Software Metrics and Design Patterns. Proceedings of the 6th European Conference on Software Maintenance and Reengineering, Budapest, Hungary, 2002, 108-115.

[Hal97] M.H. Halstead, Elements of Software Science, Elsevier, North-Holland, New York, 1977.

[Hen96] Henderson-Sellers, B., Object-oriented metrics: measures of complexity, Prentice-Hall, pp.142-147, 1996.

[HS90] S. Henry and C. Selig, "Predicting source-code complexity at the design stage," IEEE Software, vol. 7 (2), 1990.

[ISO9126] ISO/IEC, "ISO/IEC 9126. Information technology - Software product evaluation - Quality characteristics and guidelines for their use" International Standards Organization, Geneva 1991.

[KB04] Cem Kaner and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? In Proceedings of 10th International Software Metrics Symposium METRICS 2004, 2004.

[Mar94] Robert C. Martin, Object Oriented Design Quality Metrics an Analysis of Dependencies, <http://www.objectmentor.com/resources/articles/oodmetrc.pdf>, 2004-06-04.

[Mar02] R. Marinescu. Measurement and Quality in Object-Oriented Design. PhD thesis, Politehnica University of Timisoara, October 2002.

[McC76]. T.J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, October 1976, pp. 243 - 245.

[MDP07] NASA IV&V Facility Metrics Data Program data repository. This repository contains software metrics and the associated error data at the function/method level. <http://mdp.ivv.nasa.gov/index.html>

[MM85] Y. Miyazaki, K. Mori. "COCOMO evaluation and tailoring", Proceedings of the International Conference on Software Engineering (ICSE '85), pgs. 292-299

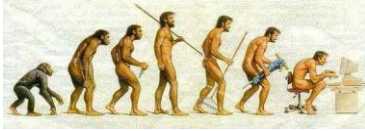
[MR04] R. Marinescu and D. Ratiu. Quantifying the Quality of Object-Oriented Design: the Factor-Strategy Model. In 11<sup>th</sup> Working Conference on Reverse Engineering, Deft, November 2004. IEEE Computer.

[OSG05] The OSGi Alliance, "About the OSGi platform", Technical Whitepaper, 2005.

[Pre95] W. Pree, "Design Patterns for Object-Oriented Software Development", Addison-Wesley, 1995.

[SMY74] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," IBM Systems Journal, Vol. 13 (No. 2, 1974), pp. 115-139.

[Sne97] Sneed, H. M. (1997). "Measuring the performance of a software maintenance department. In Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR 1997), pages 119–127.



[Sta84] Standish, T. "An essay on software reuse". IEEE Transactions on Software Engineering SE-10, 1984.

[WO95] Welker, Kurt D. & Oman, Paul W. "Software Maintainability Metrics Models in Practice." Crosstalk, Journal of Defense Software Engineering 8, 11 (November/December 1995): 19-23.

[ZK02] Zou, Y., Kontogiannis, K. "Migration to Object Oriented Platforms: A State Transformation Approach" ICSM 2002.

[ZSG79] Zelkowitz, M., Shaw, A. & Gannon, J. "Principles of Software Engineering and Design". Prentice-Hall, 1979.