EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Improving software maintainability
a case study

Sonnenberg, C.J.

Award date:
2006

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER’S THESIS

Improving software maintainability

A case study

by
C.J. Sonnenberg

Supervisor : dr. L.J.A.M. Somers

Advisor : ir. H.M.J.M. Dortmans

Eindhoven, October 2005

A school should not be a preparation for life.
A school should be life.

Elbert Green Hubbard (1856-1915)

Summary

Evolution keeps software fit for use and can be seen as the overlap between the maintenance activities and the development of new features on top of an existing code base [Mansurov 02]. An undesirable side effect of software evolution is deterioration. Deterioration effects the maintainability of the software.

The improvement of the design of a deteriorated software module is the subject of this thesis. The deteriorated module is part of a controller. A controller can be regarded as a piece of software that forms a logistic junction of data flows in hardcopy devices and systems. This report refers to this module as the job and document module (JDM), because it is concerned with job and document data.

Since its first release developers have adapted and extended JDM's functionality. More over JDM is intended to be generic and developed to suit multiple hardcopy devices for different market segments. Despite this evolution, so far no explicit effort has been made to control the growth of JDM's complexity.

The redesign of JDM should address all identified and future requirements and convincingly improve on the non-functional requirement maintainability with respect to the original design. This redevelopment is complicated, because the existing code base imposes additional engineering constraints and existing design decisions.

Suitable workflows, workflow details and activities from the Rational Unified Process (RUP) are selected and adapted to address this redevelopment. To stress maintainability, the quality framework ISO 9126 (qualitatively) and object-oriented metrics (quantitatively) complement the selected elements from RUP.

JDM's source code was measured with a broad set of metrics. The results showed that deep inheritance, tight coupling between the classes and high complexity characterise JDM. The trends that we revealed point out that principally in its evolution JDM has been increasing its complexity and size without changing the structure of its model. This analysis shows that in the area of maintainability, problems with respect to analysability can be expected in understanding the dynamic aspects, tracking and tracing bugs. Stability is compromised, because bugs are easily introduced. Testability decreased, because the effort for among others path coverage increases with the complexity.

A review of the requirement documentation showed that the non-functional requirements that have been refined and strengthened over time mainly contributed to the deterioration of JDM. In contrast, the identified initial functional requirements appeared to be quite stable in number and in detail. From the documentation and interviews with the stakeholders, we distilled that the frictions with the original design is concerned with:

- the analysability of the mechanisms,
- the interfaces that lack abstraction from the implementation,
- the editing of jobs, especially when it involves changes to the object structure of the model and
- the performance (including streaming, notification, locking mechanism and foot print).

JDM's original design model was the start point of our redesign. The use case models were derived from JDM's documentation and were adapted. Now, the use cases show a higher level of collaboration between the actors and JDM and the collaboration adheres to the tasks that actors have to perform.

The logical view was retrieved and reviewed. We reused the design classes of JDM and defined three principal operations:

- allow by design only meaningful/used object structures,
- methods operate only on the attributes and objects that are directly associated to the invoked object
- base classes cover the interfaces of their derived classes

These operations turn the model around and improve on the analysability, because these operations have effect on the Depth of Inheritance Tree, Coupling Between Objects and McCabe's Cyclomatic Number. We further concentrated on the interface to the model. In our case, the design mechanisms provided a start point for appropriate interfaces. The Builder pattern [Gamma 95] is linked to the Composite and is useful to access and create the job structures. The interface for the Workers is based

on the Producer/Consumer design mechanism, because they consume pages, transform them and produce pages for the next worker inline. Not readily available was a mechanisms to stream composite like structures and to model information concerning the structure of jobs and documents. For this, the similarity with abstract syntax trees was considered.

In retrospect, we conclude that JDM was the right place to start to consider improving the maintainability of the controller. Because the number of dependencies has decreased considerably, behaviour and data structure have been separated and functionality is expressed only by means of attributes, our redesign, in general, allows for a much simpler implementation of the features as opposed to the original design of JDM.

We observed that object-oriented metrics, like depth of inheritance and coupling between objects, are helpful to identify the 'hot spots' within the software. Metrics can form a base for future guidance of non-function requirements, especially maintainability. This can help to preserve and monitor architectural integrity in the future.

Acknowledgement

This thesis marks a period of six years, in which I pursued, in part-time, my master degree in Computer Science at the Eindhoven University of Technology.

I started this course to gain a better understanding of the fundamental principals of this specialism in order to create a solid base for my functioning on the longer term. If I evaluate this motivation, I feel that this course not only contributed to my learning process, but also inspired me to view (scientific) problems from different perspectives.

At times, attending and especially accomplishing this course did not come easy to me. Dividing my attention, time and energy over my study, my daytime job and my family life intensified this experience. In the light of their patience and their contribution to this process and this thesis, I like to express my gratitude to my colleagues, friends and family.

In particular, I thank Klaas Kuin. Eight years ago, Klaas offered me a position in his department at Océ-Technologies B.V. and the opportunity to educate myself. Looking back I am very glad that we have met. In this period, I have grown professionally as well as personally and Klaas has contributed to this process with his personal interest and support for me.

Eric Dortman is a very experienced colleague, who has an excellent sense for elegance in software design and architecture. His character profile made him the right person to supervise this specific assignment. Thank you, Eric, for your guidance concerning the content of this assignment. I am also grateful to Lou Somers for his precise and helpful guidance with writing this thesis. In the past half year, I had some personal discussions with Wim Couwenberg, which helped me to finish this thesis. Wim Couwenberg has been a coach to me for quite some time now. I really appreciate that.

I am grateful to many of my colleagues and to other people in my environment, especially Rob van de Tillaart, Rolf Eisenberg, Ron Dielhof, Leon Maessen, Thomas van den Broek and Douwe Bootsma, who helped me discuss the results, supplied me with feedback, the coffee breaks and lunch walks. Thank you, Bas Graaf and Marco Lormans at Delft University of Technology, for sharing articles and information with me.

Finally I stop by the persons that are most dear to me. The on going love and support of my mother, father, brother and sister gives me confidence. Being able to share experiences and events with friends is very valuable to me. Annemarije, these past years in which I attended this course have been the most intense. In this period, we got married, bought a house, moved and our sons Niek and Lucas were born. Thank you for being my partner in life.

Kees Jan Sonnenberg

Table of Contents

1 Introduction *1*
2 Job and Document Module *3*
2.1 Collaboration 3
2.2 Functionality 3
2.3 Design 4
2.4 Problem Definition 4
3 Software Deterioration *6*
4 Working Method *8*
4.1 Analysis workflow 10
4.2 Redesign Workflow 11
5 Analysis *13*
6 Redesign *24*
7 Conclusions and Recommendations *30*
8 Abbreviations *32*
9 References *33*
Appendices *35*

Table of Figures

Figure 1 System outline.....1
Figure 2 Black board architecture.....3
Figure 3 subdivision of a sequence of pages.....4
Figure 4 ISO/IEC 9126 quality model.....9
Figure 5 Analysis Workflow.....10
Figure 6 Redesign Workflow.....11
Figure 7 Trend in size and complexity of JDM.....13
Figure 9 Percentage MVG for the modules on the black board.....15
Figure 10 Original Class Diagram JDM.....17
Figure 11 Fishbone diagram of JDM's complexity.....22
Figure 12 Use case 'Create Global Job Description'.....24
Figure 13 Use Case 'Split Global Job Description'.....25
Figure 14 Use Case 'Process Job'.....25
Figure 15 Class Diagram of the refactored JDM.....26
Figure 16 Client interface.....27
Figure 17 Sharing.....27
Figure 18 A tree container for Producer/Consumer.....29

1 Introduction

The reuse of software modules in future systems and the development of product families, which are based on standard platform architectures, ought to decrease the development cost and lead-times. In reality, the continuous extension of software tends to increase its complexity, which results in the opposite effect. Effort is needed to reduce the complexity and extend the lifetime of the evolving software. In this recovering process, maintainability is the key quality attribute. If the architecture and the design of software meet this quality attribute, it improves the responsiveness and reduces the cost of the implementation of new requirements.

This thesis deals with the improvement of the design of a deteriorated software module. This module forms a crucial part of a controller of Océ. This document reports on the symptoms of the deterioration, a method to address maintainability and the redesign of the module. These results provide an improvement direction for engineers, designers, architects and others on the planning, redesign, implementation and usage of the module. The investigation has been executed simultaneously with the development of the controller. The proposed alternative design mechanisms have been prototyped, but have not been committed to the actual code base of the controller.

The previous generation of document production systems was one of analogue devices. These analogue devices, copiers, were able to reproduce paper documents e.g. memos, reports, letters etc. Since the introduction of digital technology in these devices, they have evolved and support now multiple functions. These multifunction devices can copy, but also scan and print. The controller is the component that enables these functions, because it separates the machine's components and controls the document flow through the system.

A typical Océ system is displayed in Figure 1. In this figure, the controller connects the system's scanner, operator console, printer and the network. The controller is displayed as a separate server, but this box can be integrated into one of the other system components (e.g. printer). It can be regarded as a piece of software, which forms a logistic junction of data flows in hardcopy devices and systems. Via the controller instructions and data are routed to these system components. With this system

- paper documents can be scanned and the digital documents can be routed to the network
- digital documents can arrive via the network and can be printed and
- paper documents can be reproduced via its scanner and printer.

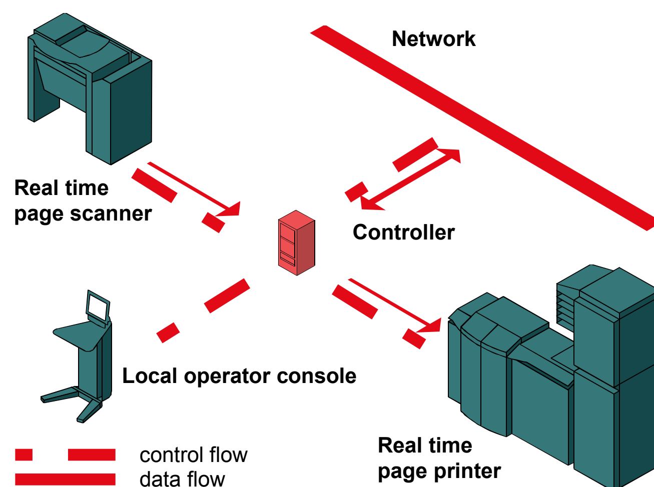

Figure 1 System outline

The controller under investigation is a complex piece of software on top of Microsoft Windows 2000, which runs on standard (PC) hardware. The controller's functionality includes rasterisation, image processing, user interface, network connectivity, scheduling, performance optimisation, job persistency, configuration- and power management, error recovery, service assistance and accounting.

The set-up of the report is as follows. The next chapter formulates the context and problem of the module more closely. Software deterioration is the subject of chapter 3. Then chapter 4 gives an outline of the methodology we used to execute the assignment. The analysis and redesign results are explained respectively in chapter 5 and 6. Finally chapter 7 expresses the conclusions and recommendations gathered during this assignment.

2 Job and Document Module

The module that we consider is called the job and document module or JDM for short. This chapter elaborates on its collaboration with the other modules in the architecture, its functionality, its design rationale and the problems that are associated with JDM.

2.1 Collaboration

The controller is a platform product, because it is part of multifunction, scanner only and printer only devices for different market segments. The demands on the architecture are strong. The controller should support not only the continuous extension of its functionality, but should also enable the proper allocation of the generic and specific functionality. The latter is important in order to be able to support multiple systems.

This controller has been built from scratch. Its architecture benefits from the experiences gained with its predecessors. These experiences and the identified requirements for this controller have lead to the 'black board' archetype [Clements 01]. The black board stores centrally information that needs to be shared or communicated between modules. The information on the black board is kept consistent by the Publish/Subscriber pattern [Buschmann 96]; the modules are notified when information on which they are subscribed is changed. The information that resides on the black board is structured into generic models that are encapsulated in modules. These generic modules are reused in all systems. The models specify "What to do" and "When to do it", but the how-question is deferred to the modules that use the model. This enables the implementation of specific behaviour of individual systems.

JDM is one of these generic, reused modules. Other modules on the black board relate to among others job queues, capabilities and available resources. JDM is in Figure 2 schematic denoted as a module on the black board with its primary users: the clients, workers and the splitter. The client modules receive a request via the network or local user interface and create a global job description. Like the name suggests, the splitter breaks down this global description into multiple descriptions for sub jobs that can be executed by the available workers. The workers interface with, for example, a scan or print engine or the Raster Image Processor (RIP) and execute the document transformations.

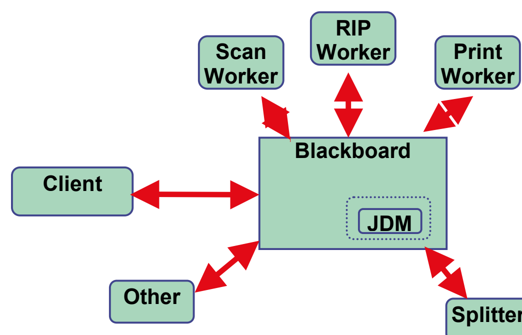

Figure 2 Black board architecture

2.2 Functionality

A simple example of a job that is processed by a multifunction device is the reproduction of an one-sided, paper document. This is called a copy job. Besides the copy job, print and scan jobs are the other supported basic job types. These jobs become more complicated to model if they include the generation of special pages (e.g. banner pages, insertions of tabs), reordering of the document (e.g. booklet, reverse printing, 2-up) and other features like subset stapling, mixed media, selection of paper trays, scaling or shifting margins.

A job is expressed by means of a sequence of (intermediate) documents. This is explained in detail 2.3 Design. JDM encapsulates a data structure with which jobs in all their states can be described. On top of this data structure, JDM provides services for interprocess communication, change notification and (exclusive) access to the data structure.

2.3 Design

The challenge in the design of JDM is to be able to represent all kinds of (future) jobs by means of one data structure. JDM's design represents jobs as a sequence of their input, intermediate and output document descriptions. This principle is referred to as the source-target model. The tasks or sub jobs involved in the transformation of one document (source document) in the next (target document) can be derived from the differences between the two documents descriptions. It is up to a number of modules, which we call workers, how the document transformation is performed.

For example in Figure 3, a print job is fully specified by its source document, a PostScript document, its intermediate document, a bitmap document, and its target document, a paper document. The RIP Worker executes the first sub job, because this worker is able to convert a PostScript document into a bitmap document. The second involves the print worker, because it is able to print the images of the bitmap document on paper sheets.

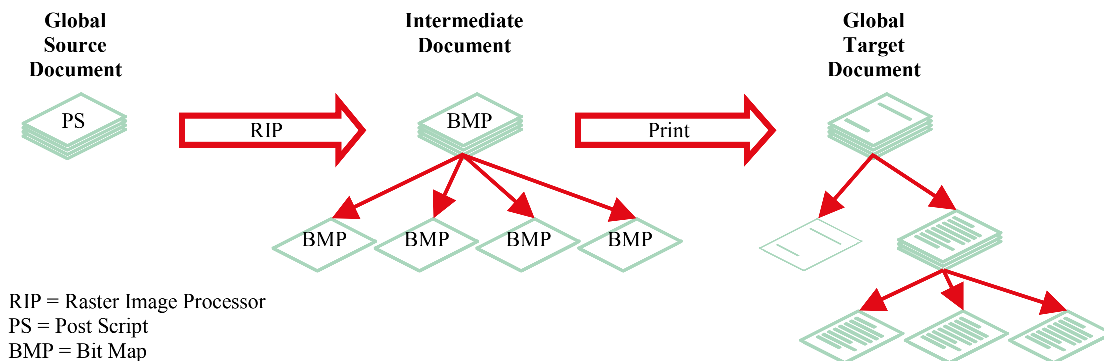

Figure 3 subdivision of a sequence of pages

In JDM, the representation of the documents is generic and flexible to be able to express all kinds of jobs. A simple document is regarded as a linear sequence of carriers. A carrier is an elementary unit that is common to all types of documents known in the system (e.g. paper sheet or bitmap file). On its turn a carriers may have one or more frames. E.g. a paper-based document can have for each sheet two frames; for each side of the sheet one. Finally each frame has an image.

A document can be subdivided in subdocuments to model more complex tasks. These sub documents preserve the order of the carriers and usually represent a logical part of the document (e.g. banner page, cover and appendix). Typically, these subdocuments are further detailed by means of attributes like colour of the substrate and staple. Each subdocument can be viewed as a document in itself and can be subdivided until the individual carriers are encountered. For example in Figure 3, the global target document represents a four-page document. An attribute associated to a document at a certain level in the hierarchy can be staple, which means that all pages within this document are stapled together. The target document is split in two subdocuments. The first subdocument is a single sheet and can represent a front cover. Additional attributes (e.g. transparency and simplex) can detail this part of the document. The second subdocument represents the remaining sheets on, for example, standard paper.

2.4 Problem Definition

JDM evolves, because new functionality and support for new systems are added in releases to this piece of software. Due to this evolution, JDM remains satisfactory in use, but the quality of its code base degraded over time. Below, we list some prominent symptoms of the degradation of this module.

Many interfaces, which lack abstraction
The interface to JDM is more or less a direct mirror of the classes that the interface encapsulates. This resulted in an interface that provides little abstraction and a large number of interfaces relatively to the limited size (< 9 KLOC) of the module. In fact there are 15 interface classes.

High complexity of the encapsulated classes
Some classes within the module are very large; up to 84 methods. The classes within the module are tightly coupled and have a deep inheritance structure. The co-operation between the classes is hard to follow, maintain and extend.

Performance
JDM reflects the system's capabilities and is involved in every processing of documents by the multifunction device. The remote procedure calls that can be used to access the module consume quite some system resources. This performance effect is enlarged by lack of abstraction of its interface and the many accesses to the module.

Redundancy in data structure
The data structure can be hierarchical composed. However, the use of the data structure is static and comprises a fixed number of levels.

Redundancy in objects and attributes in this structure
The semantics are encapsulated in attributes as well as in the type and number of relations among objects. Moreover, some features are expressed both in the structure of objects and in attributes.

Hard to modify the data structure
Because some features are expressed in the structure among the objects, an elementary change of a feature (e.g. simplex - duplex) requires a complete change of the data structure.

Bottleneck in development process
JDM is critical in the development process, because changes or problems in this module may affect many others and the core functions of the system.

The problems sketched above serve as a starting point of our investigation. In the next chapters, we describe how we overcame these problems by analysing them from multiple viewpoints and redesigning the module.

3 Software Deterioration

Deterioration problems are not unique and are well known to both industry and the academic world. The problem of 'software deterioration' means that if a piece of software ages and is maintained, the software becomes successively harder and harder to maintain [Land 02]. Software maintenance is defined by many standards [Bass 98, ISO/IEC 9126]. One of them is the IEEE definition [IEEE 91]:

"*Software maintenance is the process of modifying the software system or component after delivery to correct faults, improve performance or other attributes or adapt to a change in environment.*"

The evolution of existing software can be seen as the overlap between the maintenance activities and the development of new features on top of an existing code base [Mansurov 02]. Lehman noticed that software systems need to evolve to remain satisfactory in use and that these adjustments tend to increase the complexity of the software. Based on these observations, he formulated in 1974 his first and second law.

Lehman's Laws

Lehman distinguishes Specifiable and Embedded type software systems. The problems that S-type programs solve can be stated formally and completely. If the program meets its specification, it is accepted. This type of software does not evolve, because a change to the specification defines a new problem. E-type software becomes part to the world it models. In contrast to S-type systems, which are solely concerned with correctness, E-type systems are characterised by the following, additional criteria [Lehman 01]:

- Systems that are actively used and embedded in a real world domain.
- Systems that are judged by the results they deliver.
- Systems that, at least for the moment, satisfy their stakeholders.
- Systems that are concerned with factors such as quality, performance, ease of use, tangibility etc.

Lehman's laws apply to E-type systems. In his first and second law, Lehman expresses the tendency of the wear out of the software design, respectively [Lehman 01]:

1. An E-type system must be continually adapted else it becomes progressively less satisfactory in use. In time, the initial assumptions on which a software system is build become invalid, because the user experience increases, the user needs and expectations change, new opportunities arise and system application expands.
2. As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it. One of the reasons of the complexity increase is that the number of potential connections and interactions between elements is proportional to the square of the number of elements [Lehman 01]. The software size grows and the number of elements increases, which makes, each time, a change upon a change more difficult.

Examples of E type system that are subject to evolution, might be applications in the area of customer service, order entry, payroll, ERP, operating systems, databases engines, etc.

Deterioration Prevention & Resolution

Prevention of deterioration starts at the (software) architecture. The architecture should address the non-functional requirement or quality attribute that is called maintainability. The awareness that maintenance is an important issue in the software architecture increases, because industry is spending more and more on this phase of the software life cycle. However, there are other quality attributes besides maintainability that determine the product quality. Performance, reliability and usability are examples of such attributes and between different attributes there may be trade-offs, for example between maintainability and performance.
Despite this awareness, thirty years of research and the many suggested approaches, it is still inevitable that a software system eventually erodes under pressure of the ever-changing requirements [Gurp 02]. So in order to expand the lifetime of an evolving software system, its complexity has to be reduced once in a while or continuously in small steps, like in refactoring.

The controller is an E-type system, because it is part of devices that are sold and are used actively in the real world. This means that Lehman's laws are applicable. The controller's functionality, including new functionality, is delivered in releases to multiple hardcopy systems. With these new and changed

functionality it remains competitive. However, these modifications increase the complexity of the software. In particular for JDM, we assume that this module requires a redesign to decrease its complexity and resolve friction of requirements with the original design, because many changes have been committed to its code base and so far little effort has been put in reducing its complexity. The reduced complexity of the redesign should improve the software maintenance process and the development of new features. Redeveloping a software module in an existing system, like JDM, is a challenging task, because it is often more difficult than developing an equivalent module in a system that is build from scratch. The reason is that an existing code base imposes additional engineering constraints and existing design decisions always need to be taken into consideration.

4 Working Method

This chapter shows how we approached the deterioration problem. [Berard] delivered the criteria that we used to select our working method. According to Berard, worthwhile engineering techniques are those which:

- can be described quantitatively,
- can be described qualitatively,
- can be used repeatedly, each time achieving similar results,
- can be taught to others within a reasonable timeframe,
- can be applied by others with a reasonable level of success,
- achieve significantly and consistently better results than either other techniques or an ad hoc approach and
- are applicable in a relatively large percentage of cases.

We selected the Rational Unified Process, which is an existing object-oriented methodology, to serve as a template for our approach. This means that RUP delivered elements that we modified to suit our purpose. The quantitative and qualitative techniques with respect to maintainability are stressed in our approach by using parts of respectively (object-oriented) metrics and a quality framework (ISO 9126). Below we explain these choices.

Rational Unified Process

Rational Unified Process (RUP) is a software engineering process. RUP integrates the Booch methodology [Booch 94], Object Modelling Technique [Rumbaugh 91] and Use Case modelling [Jacobson 92] of respectively Booch, Rumbaugh and Jacobson. The software engineering process is the process of developing a system from requirements, either new (initial development cycle) or changed (evolution cycle). In RUP, this process is decomposed into four sequential phases: inception, elaboration, construction and transition. A milestone concludes each phase respectively lifecycle objectives-, lifecycle architecture-, initial operational capability and product release milestone. Each phase can be subdivided into multiple iterations. In these iterations activities of six different workflows are executed. A workflow describes a sequence of activities, artefacts and involved roles.

With respect to the criteria of Berard, we assume that this method is applicable in a fairly large number of cases and can be taught to others quite easily, because the industry embraces RUP and its associated Unified Modelling Language (UML). In addition, one of the founders of RUP, Rumbaugh, has established the confidence that reasonable levels of success are obtained consistently and repeatedly. He stated that an object-oriented approach produces a clean well-understood design that is easier to test, maintain and extend than non-object-oriented designs, because the object classes provide a natural unit of modularity [Rumbaugh 91].

Quality frameworks

The engineering project of the controller have addressed quality by adopting the standard ISO 9126, which provides a framework for the quality characteristics (e.g. maintainability). We adopted this quality framework too, in order to discuss maintainability qualitatively. ISO 9126 belongs to the category of well-known software quality models, just like Gilb, McCall, Boehm and FURPS [Ortega 02]. Mostly, these quality models are hierarchical and distinguish at different levels quality attributes or characteristics. On the higher levels, the attributes are closer to the perception of the involved stakeholders, while the attributes on the lower levels are more easily captured by metrics. Below, the hierarchical model of ISO 9126 is shown.

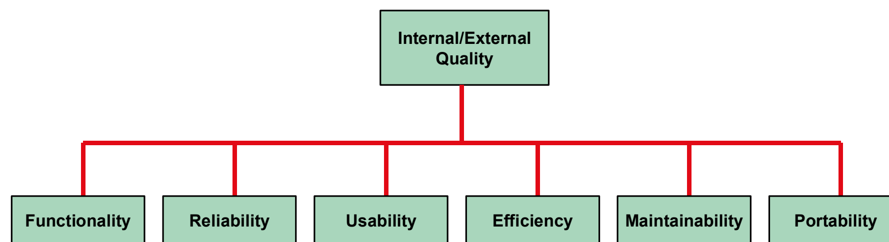

Figure 4 ISO/IEC 9126 quality model

ISO/IEC 9126 does not provide requirements for software, but it provides a framework for the evaluation of software quality. This framework is applicable to every kind of software. It defines six product quality characteristics (See the figure above). Maintainability is one of these quality characteristics and the standard's definition is:

"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."

In addition, the standard provides a suggestion of quality sub characteristics (see Table 1) [Essi-Scope, http://www.cse.dcu.ie/essiscope/].

Subcharacteristics	Definitions
Analysability	Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.
Changeability	Attributes of software that bear on the effort needed for modification, fault removal or for environmental change.
Stability	Attributes of software that bear on the risk of unexpected effect of modifications.
Testability	Attributes of software that bear on the effort needed for validating the modified software.

Table 1 Subcharacteristics of Maintainability

One common thought in most models is the use of measurements to monitor the quality and to identify and eliminate the causes of defects.

Software metrics

Software development process and software product design both can be measured individually. We focussed on product quality. Product metrics are measurements, which are applied to the products created in the process, including source code, the final executable program, analysis or design documentation [Littlefair 01].

In this assignment, we chose object-oriented metrics to address maintainability quantitatively, because the results are objective and there is a strong relationship between metrics and maintenance effort in object-oriented systems [Li 93]. Suites of object-oriented measures for code and design have been proposed by, among others, Chidamber & Kemerer [Chidamber 91] and Lorenz & Kidd [Lorenz 94]. One of the measures of a suite is often McCabe's Cyclomatic Number. This specific metric, which determines the number of independent paths through a program, is viewed as an indirect measure of maintainability [McCabe 76]. More specific, we used the object-oriented software metrics for model and class quality suggested by Lorenz [Lorenz 94] to quantify the quality of design (see Appendix A). In addition, we measured the software releases of JDM with a broad set of metrics (see Appendix A) that helped us to determine which metrics the design is most receptive to. The trends and the comparison of the measures with values provided by literature provide information that we used for analysis of the deterioration. The values of selected measures are also useful to validate the redesign.

Metrics are useful, however, some thoughts are important to keep in mind with respect to the use of software metrics:

- In general, measurement theory categorises measures into direct and indirect measures. Direct measurement of an attribute is a measure, which does not depend on the measurement of one or more other attributes. On the other hand, indirect measurement of an attribute is a measurement, which involves the measurement of one or more other attributes.
- The results of measurements are guidelines not rules; metrics give an indication of the quality of the design. Metrics can violate certain pre-set values, but there should be a good explanation why these values are exceeding (e.g. the interfaces in C++ require multiple inheritance).
- Although there are many measures of software quality, the after-the-fact measures are the most widely used.
- Product metrics are influenced by many factors; among others the domain for which the software is developed, the state of the software (e.g. prototype, first release, etc.) and previous project experiences.
- Any comparisons between the computed values of the metrics should be done only between projects that have been developed for similar requirements and objectives or have comparable solutions.
- In any development process, the acceptance of metrics will be increased if automated tools are in place that capture the results and the measurements are intuitively meaningful to persons involved in the development.

4.1 Analysis workflow

The goal of this workflow is to develop a sense of urgency, identify the problems and the direction of change. Our analysis workflow is depicted in Figure 5. It consists of the following steps:

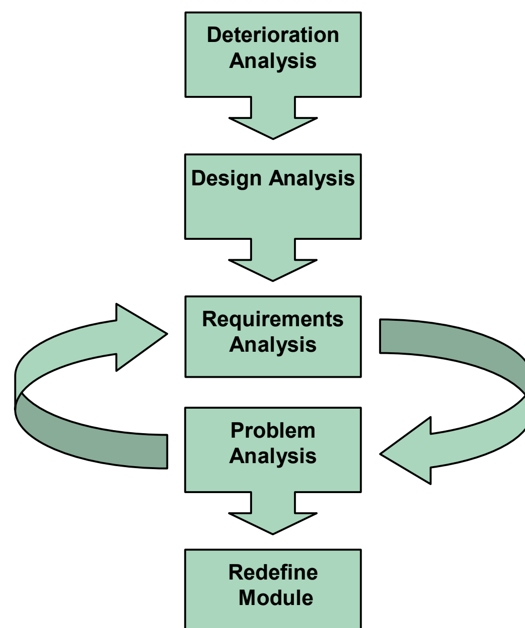

Figure 5 Analysis Workflow

Step 1: Deterioration analysis
This activity provides input to decide that the suspected software is eroding and needs to be repaired. In this case, we restricted ourselves in providing symptoms and indicators of the increasing complexity of the original software. The timing of the repair is out of the scope of the assignment, because we limit ourselves to product quality. The technique we used in this phase is:

- Use metrics to reveal trends

Step 2: Design Analysis
Its purpose is to detect any design flaws and to evaluate the architectural quality maintainability in the original software design. It also serves to identify mismatches between the requirements and the architecture, for example, over-design and unrealistic or missing requirements. Architectural flaws are

known to be the hardest to fix and the most damaging in the long run. The following steps in this activity have been executed:

- Representation-driven review
  Representation-driven review is conducted by obtaining a representation of the design, then ask questions and reason based on this representation. For this review, we have obtained from the source code the logical view or class diagram. The representations of the design of JDM provided by the project documentation were not useful, because they provide a simplified view on the design.
- Information-driven review
  In an information-driven review information (e.g. data, measurements) is established and is used for the reasoning; in other words retrieve the information and compare this information to either the requirements or some accepted reference standard. A suite of metrics and their thresholds proposed by Lorenz [Lorenz 94], which is our reference standard, is used to determine class and model quality. The measurements are executed manually.

Step 3: Requirements Analysis
Stakeholders are identified and their requests are elicited. The collected stakeholder requests are regarded as a "wish list" that will be used as primary input to defining the high-level features of the system. In addition, requirement documents are retrieved and reviewed. 'Understand stakeholder needs' is the workflow detail from RUP, which provides useful techniques for this step. The techniques that are used in this step are

- Interview stakeholders
- Review existing requirements
- Review evaluation reports

Step 4: Problem Analysis
This analysis step aims at developing a vision on how and why the deterioration took place. This vision is expressed as a set of the most critical problems to solve in the design. 'Analyse the problem' is the workflow detail from RUP that provides useful techniques for this step. The following technique is used in this activity.

- Use Fishbone diagrams to clarify the problem

Step 5: Redefine Module
In this step, we are focussed on identifying actors and the most relevant use cases more completely and expand the global non-functional requirements. No additional techniques are used.

4.2 Redesign Workflow

Our redesign workflow is depicted in Figure 6. The explanation of these activities is provided underneath this figure.

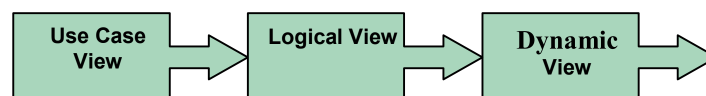

Figure 6 Redesign Workflow

Step 6: Use-Case View
This activity is used to refine use-case realisations in terms of interactions between actors, to refine requirements on the operations of design classes and to refine requirements on the operations of this subsystem and their interfaces. 'Use-Case Design' is the workflow activity from RUP, which provides useful techniques for this step. The following steps in this activity have been executed:

- Describe Persistence-related Behaviour
- Handling Error Conditions
- Handling Concurrency Control

Step 7: Logical View
Its purpose is to analyse interactions of analysis classes and to identify design model elements. 'Identify Design Elements' is the workflow activity from RUP, which provides useful techniques for this step. The following steps in this activity have been executed:

- Identify and Specify Events
- Identify Classes
- Identify Subsystem Interfaces

Step 8: Dynamic View
Its purpose is to refine the analysis mechanisms into design mechanisms based on the constraints imposed by the implementation environment. 'Identify Design Mechanisms' is the workflow activity from RUP, which provides useful techniques for this step. The following steps in this activity have been executed:

- Categorise Clients of Analysis Mechanisms
- Inventory the Implementation Mechanisms
- Map Design Mechanisms to Implementation Mechanisms
- Document Architectural Mechanisms

5 Analysis

The deterioration analysis workflow that we described in the previous chapter has been applied to JDM. The results of this process are given in this chapter.

Step 1: Deterioration Analysis

We verified that it make sense to address the maintainability of JDM. By means of the tool CCCC[1], we collected a wide variety of metrics from the source code. The metrics that this tool supports are enumerated and explained in Appendix A. Appendix B and C show the results of the measurements of the critical classes that are part of the blackboard of the controller.

By measuring the code base for different releases, we revealed the trends in the metrics. The revealed trends show that throughout its evolution the number of classes and relations among them did not grow strong for JDM. The most interesting result is displayed below. In this figure, the McCabe's Cyclomatic Number (MVG) is plotted against Lines of Code (LOC). LOC has an obvious relation to the size or complexity of a piece of code and can be calibrated for use in prediction of maintenance effort. MVG is now widely accepted as a measure for the detection of code, which is likely to be error-prone and/or difficult to maintain.

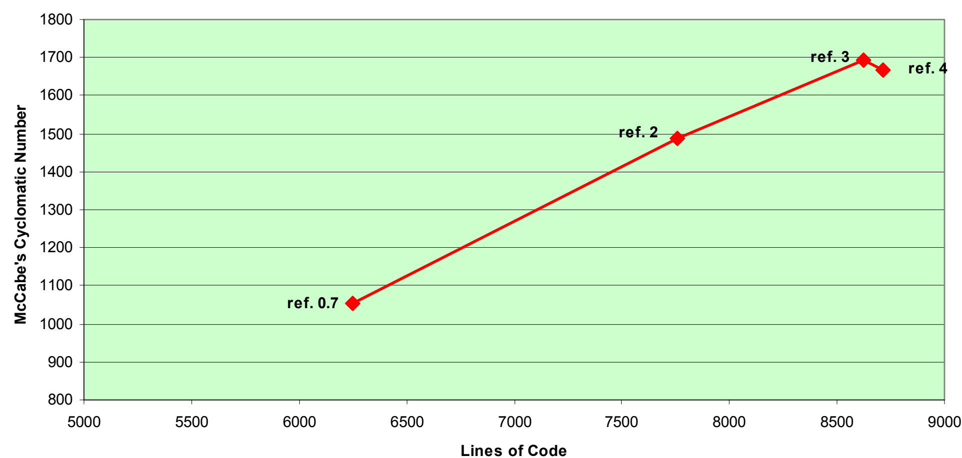

Figure 7 Trend in size and complexity of JDM

Figure 7 shows that the complexity and size of JDM increased strongly with each evolution[2]. If this trend persists, the problems associated with maintainability will increase with each new release of JDM.

CCCC provides a set of reference values and marks the values of measurements that do not comply with those pre-set values. The marked classes are suspected and might compromise the quality. The pre-set values we used, were the defaults configured in the tool. According to this reference set JDM's quality is compromised by the module's (1) deep inheritance structure, (2) tight coupling between the classes and (3) high complexity.

The use of CCCC eased the executing of this analysis, but to use these measurements quantitatively the results needed to be verified. On the one hand, the tool does not support all programming language features and on the other hand interpretation is not trivial, because the semantics get lost in the

[1] CCCC - C and C++ Code Counter – is a free software tool for measurement of source code related metrics by Tim Littlefair. The CCCC tool was developed as a testing ground for a number of ideas related to software metrics in a MSc project. The research project is described at http://www.fste.ac.cowan.edu.au/~tlittlef

[2] The last measurement (ref. 4) seems not to comply with the other measurements. It might be due to the limitations of the tool. Especially in the latest release relatively more programming constructs (declarations of exception handling, option parameters etc.) are used with which the tool can not deal.

aggregated information that the tool provides. After verification, we foresee the following consequences for the subcharacteristics of maintainability:

- Problems with respect to analysability can be expected in understanding the dynamic aspects, tracking and tracing bugs.
- Changeability decreases, because the classes are increasing their size and exchanging them requires more effort.
- Stability will decrease, because bugs are introduced easily in this highly connected module.
- Testability decreases, because the effort for a popular testing method, path coverage, increases linear with the MVG. This complexity number is increasing for JDM.

According to the above, JDM is deteriorating, but is this module the right place to start addressing the maintainability of the controller? To answer this question, we gathered with same tool metrics of the other modules on the black board (including the models TAM, CAM, REM, QSM and JDM, but also KM and OP). First, we give an impression of the size of the blackboard. The accumulated values of the measures are shown in the table below.

Metric	Tag	Overall
Number of modules[1]	NOM	407
Lines of Code	LOC	32164
McCabe's Cyclomatic Number[2]	MVG	4141
Lines of Comment	COM	32954
LOC/COM	L_C	0.976
MVG/COM	M_C	0.126
Information Flow measure (inclusive)[3]	IF4	117652
Lines of Code rejected by parser	REJ	2985

Table 2 Measures over blackboard provided by CCCC

The results of the measurements, which we took up in Appendix B en C, show that

- the tool indicates potential problems in all modules of the black board, but JDM contains the most and the most severe deviations from the default reference values.
- the suspects for QSM mainly have (1) a large interface and (2) the coupling between classes is above the threshold.
- within Common there are a few classes that draw attention. There is a base class CException that has seven children. This implies a high coupling, but the class itself is very simple in nature, because it only encapsulates error codes for exception handling. Also, another class exhibits a large number of subclasses, but it is part of the framework and shields developers from the interface technology.
- The remaining classes that are marked by the tool are rather simple classes and consequently we do not consider them as a problem with respect to maintainability.

[1] The number of modules refers to the number of types or classes.
[2] McCabe's Cyclomatic Number is a measure of the decision complexity. The strict definition of this measure is that it is the number of linearly independent routes through a directed acyclic graph, which maps the flow of control of a subprogram.
[3] Information Flow measure is a composite measure of structural complexity, calculated as the square of the product of the fan-in and fan-out of a single module.

JDM's size and complexity in comparison with the other modules on the blackboard further confirm the selection for JDM. This is shown in the figures 8 and 9 below.

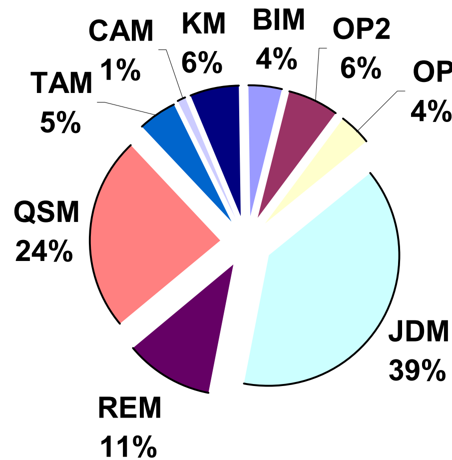

Figure 8 Percentage LOC for the modules on the black board

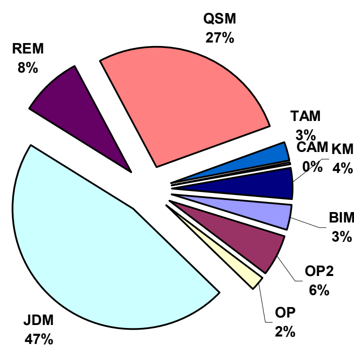

Figure 9 Percentage MVG for the modules on the black board

Step 2: Design analysis

Lorenz provided a suite of metrics that we used to conduct our information-driven review. Appendix A enumerates and explains the metrics suggested by Lorenz. These measurements are derived manually from JDM's source code (Ref. 4). In the tables below, the results of the metrics with respect to model quality and class quality are given.

Metric\Model	JDM
Hierarchy Nesting Level (HNL)	The maximum hierarchy nesting level is seven. In the class diagram in figure 10 shows five levels, but there are two more base classes above the class KMObject. Those base classes are part of the framework that, among others, provides an abstraction from the COM technology.
Multiple Inheritance (MUI)	Multiple inheritance is not used in JDM.
Global Usage (GUS)	No global variables are declared.
Friend Functions (FFU)	Friend relationships are depicted in the class diagram in figure 10 with the stereotype “friend”. JDMDocument is a friend of class JDMJob, JDMNode is a friend of JDMTemplate as well as JDMJob and CJDMTemplate is a friend of JdmIteratorDepthFirst.
Function-oriented code (FOC)	The percentage of function-oriented code is 0%. All methods belong to a certain class.
Intersubsystem relationships (ISR).	This metric is not applicable in this case.
Percentage of Commented Methods (PCM)	Average number of commented methods is > 80%. The usefulness of the comments is not taken into account; just an annotation is enough.

Table 3 Metrics guiding model quality [Lorenz 94]

Table 4 elaborates on 9 suggested metrics for class quality for the classes CJDMNode, CJDMCompositeNode, CJDMImageSequence, CJDMDocument and CJDMBitmapDocument. These classes form a chain in the hierarchy of the model (see figure 10).

Metric\Class	Node	CompositeNode	ImageSequence	Document	BitMapDocument
Number of pubic instance methods (PIM)	47	47 + 0 = 47	47 + 0 + 10 = 57	47 + 0 + 10 + 5 = 62	47 + 0 + 10 + 5 + 5 = 67
Number of instance methods (NIM)	75	5	27	23	17
Number of instance variables (NIV)	8	2	3	2	0
Number of class methods (NCM)	9	9 + 0	9 + 0 + 2	9 + 0 + 2 + 3	9 + 0 + 2 + 3 + 3
Number of methods overridden (NMO)	0	4	9	13	8
Specialisation index (SIX)	3*0/47 = 0	4*4/47 = 0.34	5*9/57 = 0.79	6*13/62 = 1,26	7*8/67 = 0.84
Percentage of commented methods (PCM)	>80%	>80%	>80%	>80%	>80%
Problem reports per class (PCR)	-	-	-	-	-
Class reuse (CRE)	No	No	No	No	No

Table 4 Metrics of class quality

The results in table 3 and 4 are compared to the thresholds that Lorenz provided. A summary of the quality is given and discussed below.

- An inheritance hierarchy that is too shallow or too deep has quality repercussions, because the deeper the inheritance tree for a class, the harder it may be to predict its behaviour. Lorenz defines 6 levels as a threshold, but the implementation of JDM shows 7 levels.
- Though exceptions can be made for the implementation of interfaces, the threshold in the literature for multiple inheritance is 0. The implementation of JDM shows no multiple inheritance.
- Global use is restricted in the implementation of JDM to a few class variables (variables that are known to all instance of the class) that are constant. Threshold in the literature is one for system anomalies.
- Several friend constructions are used in the implementation of JDM, but friend constructions break up the encapsulation. The threshold with respect to friends is zero. Friend constructions can be justified for mathematical operators or some design patterns like ‘iterator’ or performance issues, but this is not the case for JDM.
- There are no global functions used in JDM, because the implementation shows no function-oriented code. JDM satisfies this criterion, because the threshold for these functions is also 0.
- The implementation of JDM exceeds the minimum threshold for the documentation of the methods and is over 80%. Though the quality of the comments is not taken into account, this indicates that JDM is well documented.

- With respect to class quality, only five classes of JDM are inspected by means of metrics. The inspection shows that CJDMNode is a critical class within the model. The class CJDMNode has a large number of methods (75). This indicates a large responsibility of the class.

The results of measurements based on the suite of metrics proposed by Lorenz are inline with results provided by the tool CCCC. This means that though the source code is well documented, the deep inheritance structure, size and complexity of class methods compromise the analysability of JDM. The break up of the encapsulation affects the changeability of the exposed classes of JDM.

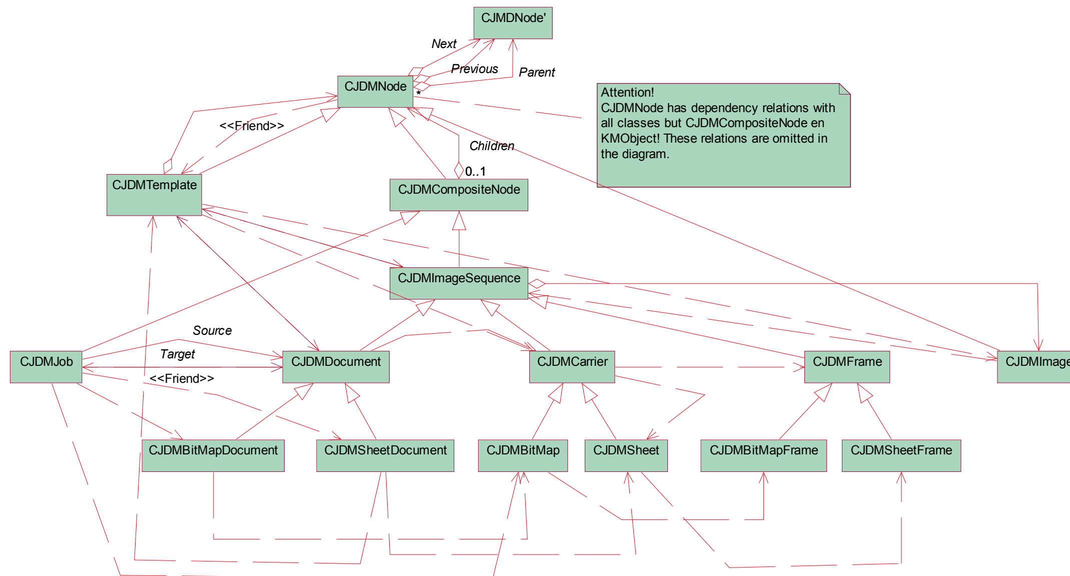

Figure 10 Original Class Diagram JDM

The following step in our design analysis is a representation-driven review. Above, JDM's class diagram with all its associations among its classes is shown. The diagram is derived via reverse engineering. In addition to the classes depicted, a number of classes (23) are used to wrap the model in COM, a Microsoft interface technology. This review provides some of the most convincing evidence of deterioration by just looking at the design:

- There is an enormous amount of dependencies among the classes. In particular, the class 'CJDMNode' knows almost all classes including its subclasses in the hierarchy. There are exceptions (e.g. the design pattern 'facade') where it is valid for a class to have a lot of dependencies.
- The ratio between the number of dependency and aggregation relations is out of balance, because it creates flexibility that is not needed. This reflects, in this case, a lack of domain knowledge in the model, because the structure of the model allows hierarchical aggregation of documents within a document, but also for example documents within a BitMap. The latter is not a valid representation.

The obvious drawback of executing the reverse engineering by hand is that it is very labour intensive. On the other hand, it provides hands on experience and feeling with the code base. In this way, details (e.g. program layout or rules) are encountered that the tools not provide in their aggregated information. We experienced that

- there are about 15 interface classes in place, but these classes directly reflect the internal classes and the model. Via this interface that provides little abstraction, the users of JDM are exposed to a large portion of the complexity of the model and make users of the module viable for internal changes to the model.
- on top of its own interfaces, JDM is wrapped with two interfaces for its users: one for its Clients and one for the Workers. These wrappings are ‘thick’ and especially the interface to the Clients provides little abstraction for the underlying model of JDM. Just like the interface to JDM itself.
- JDM has is broad technology base; for example some containers (linked list, stack) are implemented by the project, others are used from the Standard Template Library.
- the composite pattern is not properly implemented.

This representation driven review, which is based on a from source code reengineered class diagram, clearly confirms the deep inheritance and tight coupling between the classes of the model. In addition, this representation shows dependencies that clearly violate object-oriented principles and the superfluous flexibility of the model. The latter one implies that most object structures that can be build from the model have no meaning in the domain the controller is active in.

Step 3: Requirements Analysis

The start point of this analysis is JDM's “inception requirement document”. This document has been created at the start of the engineering activity in November 1999. The latest version of this document (1.2) lists forty-one explicitly numbered requirements. According to the documentation, this list contains seven future requirements. Though the formulation of the requirements take on different forms, we have categorised these requirements in functional and non-functional requirements (see respectively Table 5 and Table 6).

Number	Requirement
1	The JDM document description must support both physical (paper) documents and binary documents (e.g. PDL or bitmap files) as well.
2	It must be possible to describe documents with an unknown number of sheets.
3	The progress of a job is stored in the JDM.
4	All settings done at a Job & Document description must be restorable.
5	JDM must support simplex original on the glass platen as source.
6	JDM must support simplex original in the ADF as source.
7	JDM must support duplex original in the ADF as source.
8	JDM must support binary document (e.g. PDL) as source.
9	JDM must support two up document as source.
10	JDM must support same up document as source.
11	JDM must support booklet document as source.
12	JDM must support simplex document as target.
13	JDM must support duplex document as target.
14	JDM must support binary document (scan) as target.
15	JDM must support two up as target.
16	JDM must support same up as target.
17	JDM must support booklet as target.
18	For each combination of source and target, the run-length of a job can be specified.
19	It must be possible to express enlargement or reduction of the original images for copy jobs.
20	The JDM must support page programming.
21	The JDM must support set composition.
22	The JDM must support means to express collate on/off.
23	The JDM must provide means to indicate that a job is stopped due to some kind of error situation.
24	Settings can be overwritten with new settings at any moment.
25	The JDM must provide means to navigate through a Job & Document tree in two directions: top-down and bottom-up.
26	More sophisticated navigation like iteration (e.g. get next sheet, Get next image etc.) must also be provided for release 1.0 of the repro controller (req. 6).
27	The JDM offers means to navigate from source image to target image and vice versa.

Table 5 Functional requirements of JDM

Number	Characteristic	Requirement
28	Reliability	Job & document models that are built using JDM may be made persistent.
29	Efficiency	To prevent problems regarding persistency, the memory usage of JDM must be as low as possible.
30	Efficiency	JDM does not interact directly with the BMM. Instead, the JDM uses the Bitmap Model (BIM) as proxy for the BMM.
31	Efficiency	The performance of the JDM code may never have impact on the normal execution of a copy or print job.
32	Efficiency	The JDM must provide means to inform interested clients about changes that are initiated by other clients.
33	Reliability	The JDM must provide means to protect the objects in a job & document description for concurrent access.
34	Usability	It must be possible to get the progress of a job out of the job/document description.
35	Usability	The JDM must provide means to manage the lifetime of JDM objects.
36	Usability	The JDM provides some kind of generic interface through which all attributes can be set and retrieved.
37	Usability	All attributes that are part of the JDM interface will get a unique ID.
38	Usability	The JDM provides means to make copies of entire branches of an existing job & document description.
39	Changeability	Within the repro controller, each original copy-job description can be divided in several worker-jobs, one for each worker that is involved in the work to be done.
40	Changeability	Each job contains a description of one (or more) source document and one target document.
41	Testability	The code that is produced for the JDM module must be testable by automatic test procedures in order to perform regression tests frequently.

Table 6 Non functional requirements of JDM

In order to investigate the stability, we traced these requirements back to those identified in the phase prior to the start of the engineering activity. This prior phase that we call development investigated the requirements and design mechanism behind JDM. We have studied the documents that have been produced in this development phase. From these documents, we manually distilled 45 requirements. We compared both set of requirements and concluded that

- most requirements of JDM do trace back to those identified in development.
- the identified requirements in development form a superset of those in engineering. Table 7 lists the requirements identified in development that do not appear in the inception requirement document.
- the requirements for the interface, especially navigation, for the actors of JDM deviate substantially. Development proposed to shield its actors from the complexity of the model, while engineering proposed maximum flexibility by exposing a large part of the model.

Number	Characteristic	Requirement
1	Functional	Covers
2	Functional	Margin shift
3	Functional	Bundled documents
4	Functional	Booklets having multiple sections
5	Functional	Multi-page enlargement
6	Functional	Same up booklets
7	Testability	Printing status documents
8	Changeability	Flexible system easily expanded in later releases
9	Efficiency	Reuse for other projects
10	Analysability	Simplicity
11	Usability	Mapping with scan engine protocol
12	Usability	Mapping with print engine protocol
13	Usability	Mapping of UI concepts
14	Usability	Mapping CSL interface concepts
15	Usability	Interface to the jobs and documents to prevent workers navigate themselves

Table 7 Requirements identified in development that were not transferred to engineering

Even more of interest for maintainability is the stability of the requirements since the creation of the inception requirement document. The document itself was last updated in January 2000 and the ‘future’ requirements are currently addressed in new releases of the controller. However, since then changes are reported on in memos and reports. This means that since its creation the requirement document has hardly been maintained. Table 8 shows reported changes in the requirements of JDM that we found in over 20 reports and more then 10 memos.

Number	Characteristic	Requirement
1	Functional	The editing of active jobs and jobs in the request list concern: - Number of copies/Run length - Plexity: simplex/duplex - Staple: No staple/1 staple/2 staples - Paper tray: tray 1 – 4, auto (only when paper in specific tray has same size - PaperType: Normal, company paper, MyPaper - Cover: no, front, back, both Before edited also: - document type: portrait book, portrait calendar, landscape book, landscape calendar - Margin shift front x1 mm back x2 mm
2	Functional	The Océ VarioPrint 2090 and the Océ VarioPrint 2100 require covers alike the Océ 3165
3	Functional	Context of an active job will be lost when it is moved to the mailbox.
5	Functional	Active jobs need be stopped before they can be edited via the mailbox
6	Efficiency	The mechanism should improve the print speed during the submission of multiple jobs from the mailbox
7	Efficiency	The mechanism should decrease the time needed to remove the jobs form the user interface
8	Changeability	It should be easy to remove this mechanism once the commands to handle multiple job submission and deletion.
9	Changeability	To allow client-wise introduction, it should be possible to disable the postponed-deletion-mechanism
10	Reliability	Changes in the request list and mailbox are the main triggers to do a persistency action.
12	Reliability	All submitting entities ensure the submission of correct and complete job specifications.
13	Reliability	The SI only checks for start- and run time contradictions
14	Functional	Feedback to the submitting entity is not in order.
15	Stability	The behaviour after an illegal command is subject to predefined rule, which have to be specified elsewhere.
16	Functional	Non active jobs need to be suspended in some way, so that they can not be started and being edited at the same time.

Table 8 Changed requirements of JDM

Table 8 shows that since the start new, but mostly refined requirements are documented. So far the requirement analysis shows that the requirements trace back quite well and the initial requirements seem to be stable, but not specific enough.

Besides requirement documents, three evaluation reports have been appeared. A developer and two engineers conducted these evaluations. They concluded among others that

- it takes quite some time for a person to understand how JDM works.
- JDM is expensive in terms of development and maintenance.
- JDM is very flexible though most of that power is not needed.
- the object structure that represents a job contains loops.
- there are to types of pointers. Depending on the context pointers must be smart or not.
- once specified it is very difficult to modify a job, since it may require changes to the job structure.
- modelling functionality in an object structure cannot be unspecified (e.g. plenty)
- JDM does not separate the interface from the implementation. This means that evolutions of JDM have impact on all users.
- extra notification or improving the locking mechanism is needed to prevent from hampering the print process.
- both memory footprint and navigation overhead are problem areas for the support of large jobs in the controller.
- streaming was one of the major reasons to start with this new controller in the first place, the support for streaming is, until now, only Print-while-Scan. Print-while-Rip and cleanup-while-print functionality are recently implemented, but this hold not for print-while-rip-while spool.

The nature of the assignment, improving maintainability, explains our special interest in the needs of the direct stakeholders. We discern the following stakeholders: the person in charge, designers, developers and maintainers of JDM. In addition, there are architects of the controller and the users of JDM (the engineers of the Clients, Workers and the Splitter). In total, we conducted over fourteen interviews to (1) supplement the documentation, because it does not digress on all details that are

essential to understand the system and (2) identify major needs of the stakeholders (see Table 9 Stakeholders' major needs). The outcome of the interviews is summarised in the table below.

Stakeholder	Number	Need
person in charge of the module	1	Persistency
	2	Performance
	3	Footprint
	4	Predictability of development
designer of the module	5	Stability; express with a few attributes and the object structure many functions
	6	Extensible model
developer and maintainer of the module	7	Small technical basis
	8	Workable design principles from an engineering perspective
architects of the controller	9	Prevent information caching on multiple levels with a slightly different representation
developers of client modules	10	Small interface
	11	hidden from navigation though the model
	12	Less viable for changes

Table 9 Stakeholders' major needs

For the interviews, a checklist with open-ended questions was prepared. The checklist was derived from a Microsoft Word template provided by the RUP documentation. However, this checklist was seldom used directly. The experience was that after a short introduction of the assignment associations of the interviewee provided the most relevant information. In retrospect, we feel that reading the documentation in combination with conducting interviews helped us to understand the system much faster, because then the important information in the reports can be better distinguished and understood

Step 4: Problem Analysis

Initially, the requirements for JDM were identified by development. Engineering took over the lead from development and documented requirements that form a subset of those identified in development. For the interfaces to JDM, especially with respect navigation through model, engineering took their own approach.

Since the start of engineering, some new, but mostly refined requirements with respect to performance, editing and persistency are documented in memos and reports. This means that the initial requirements seem to be quite stable, but not precise enough. These refinements and additions during the engineering process form the base of the deterioration of JDM and reveal friction with the early design decisions. The problems that we have identified are inline with the previous evaluation reports and the results from the interviews with direct stakeholders. We combined the identified problems and distilled three major topics that needs to be dealt with in our redesign:

- JDM is pretty complex, because it takes quite some time to understand how it works. Moreover, JDM exports most of its complexity, because it does not separate its interface from its implementation. The consequence is that it is expensive in terms of development and maintenance, because evolutions of JDM have impact on all users. The aspects that contribute to its complexity are placed in a fishbone diagram, see below.
- A job might not be completely specified before execution and modifications to a job are difficult, especially when it concerns changes to the object structure of the job.
- The support for increasingly larger jobs in the controller in the sense of a smaller foot print and better performance (including support for streaming, fine grained notification or improving the locking mechanism).

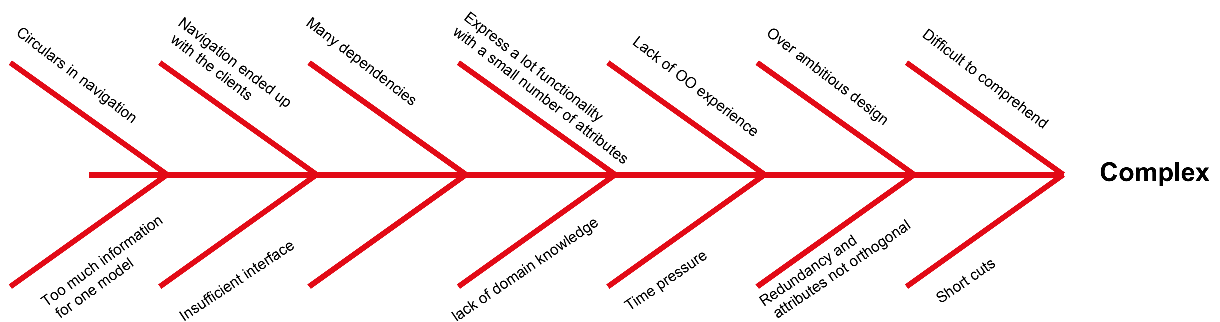

Figure 11 Fishbone diagram of JDM's complexity

Step 5: Redefine JDM

The redesign should not only satisfy the functional requirements currently implemented in JDM and solve the friction of some requirements, but also satisfy additional requirements that improve on maintainability.

JDM should be able to express by means of the source-target model:

- Print Jobs (including mixed PDL jobs PCL, PDF, PS, IPDS …)
- Scan Jobs
- Copy Jobs

Even more, the design should be able to express the following document structures:

- Banner page
- Covers (Non, Front, Back, Both)
- Plexity
- Same up
- N-up
- Booklet
- Set (sub sets)
- Concatenation
- Reverse Order

and the following document properties:

- Margins
- Tumbling
- Orientation (LEF/SEF)
- Eject boundaries
- Erase margins

In addition to the specification of jobs and theirs documents, JDM should have the following services and properties:

- Formulation of jobs
- Job persistency
- Progress information (#pages scanned or printed)
- Editing of jobs (settings can change between arrival and output, data number of image can change e.g. banner page)
- Contradictions in settings must be detected/prevented
- Unknown run length
- Limit impact of evolution
- Job can be put on hold and resumed
- Reproduce original user level information
- Page programming
- Print-while-RIP/Print while Rip while Spool
- Default values should distinguished from values that have been set (in order to be able to override them later on by postscript settings).

The redesign should resolve the friction due to the increasing size of the jobs and due to modifying job specification. But most of all the redesign should address the complexity by means of the following actions:

1. Introduce interfaces to shield clients from the complexity of the data model
2. Remove superfluous flexibility and introduce domain knowledge
3. Remove redundancy of information in data structure
4. Remove replication of data structure
5. Simplify streaming behaviour
6. Reduce the number of interfaces
7. Reduce the number of relations among objects
8. Lower complexity by separating behaviour from the data structure

6 Redesign

Following the analysis of the previous chapter, this chapter specifies the important aspect of the redesign by providing three design views according the four-plus-one approach of Kruchten [Kruchten 95].

Step 6: Use Cases

The use cases describe in normal language the flow of events between the actors and the module. The use cases not only describe the main flow, but also all the exceptions that can occur during the interaction. In the case of JDM, there are three main use cases: (A) create global job description, (B) split global job description and (C) Process Job.

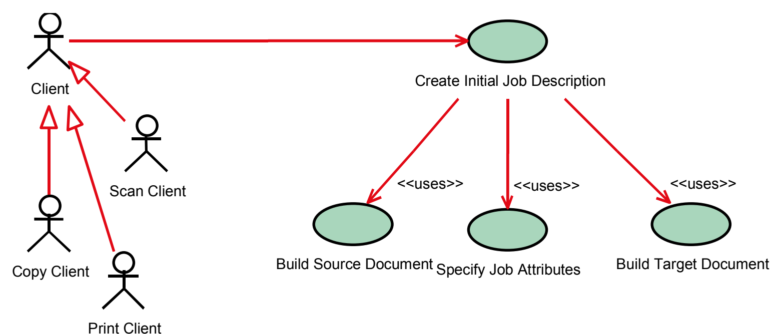

Figure 12 Use case 'Create Global Job Description'

A. *Create Global Job Description*

The actors in the use case ‘create job’ are the modules that initiate the execution of a job. They define the job. More specific, they specify a print, scan or copy job. For example the module that receives the PDL stream via LPR, which is an ad hoc standard to send a job to the spooler, from the printer driver or the user interface specify respectively a print and a copy job.

The responsibilities of the actors are to specify a job specification for scanning, printing or copying that does not contain any inconsistencies (e.g. duplex for a target document of a scan job). This specification is a description of the source and the target document.

The actor creates a new job and defines the job level attributes. Then the actor opens the source document and specifies the document level attributes. If needed, the actor opens one or more subdocuments and specifies on each level the accompanying attributes. Some subdocuments are offered in the form of templates, for example, cover or banner page. These actions are repeated for the target document.

If the objects are not created for example if the system is out of memory, the actor is informed. If an attribute is specified that does not comply with the type of document (e.g. erase margin on the target document of scan job), the client is informed.

B. *Split Global Job Description*

The actor in the use case ‘Split Global Job Description’, the Splitter, is the module that identifies the subjobs in a global job description and distributes tasks over the subjobs. For example, the splitter decomposes a copy job into a scan and print subjob.

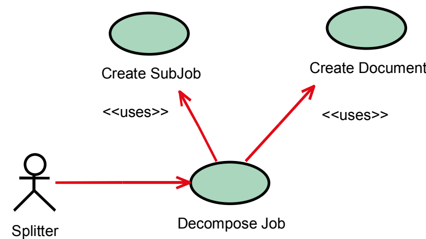

Figure 13 Use Case 'Split Global Job Description'

Based on the capabilities of the actors that are involved in executing the job, the responsibility of the splitter is to produce from a global job specification, a full job specification. The produced specification distributes the tasks over the actors.

The splitter creates new subjobs and their intermediate documents. The intermediate documents are constructed in a way that valid subjobs are created and the execution of the subjobs results in fulfilling the global job.

If the subjobs and intermediate document are not created, the Splitter is informed, for example, out of memory.

C. *Process Job*

The actors in the use case 'Process Job' are the modules that execute the job's document transformations. The main responsibilities of the actors (Scan, Print, Image and RIP Worker) concern transformation from paper to bitmap, bitmap to paper, bitmap to bitmap and PDL to bitmap. In addition, the actors for example the Scan worker applies functions like erase margin and zoom.

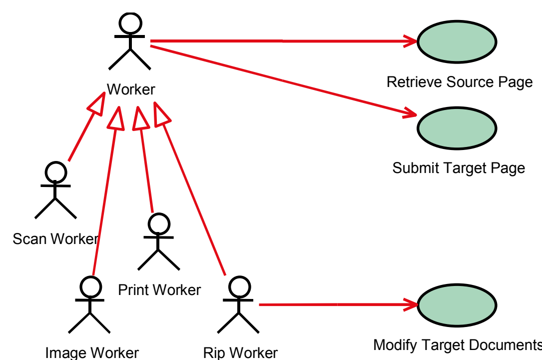

Figure 14 Use Case 'Process Job'

In a transformation, the actor looks at what page it supposed to produce and then looks what is available in the source document to produce this page. For this purpose, the actors retrieve the attributes for a target page and the corresponding source page and its settings. Then he transforms the source page in the required target page and submits the latter.
From this difference in attributes, the worker can determine the tasks that need to be executed. E.g. for a Scan Worker, the target attributes might specify the required frame size and the source page describes frame size of the original page. From the difference the zoom factor can be calculated. Additional attributes on the source page, like duplex, AFD/Glass Plate, erase margin, guide the scanning tasks. Another example is the Print Worker. The target document is subdivided into multiple subdocuments representing, for example, eject boundary.

There are exceptions to the flow. For example in the case of a banner page, no corresponding source page is available. Another exception is that the RIP worker might discover attributes that are embedded in the PDL stream. These attributes might extend or partly override the current job description.

Step 7: Logical View

The logical view or class diagram of the redesign is based on the concepts of the original design. The redesign follows from a sequence of transformations of the original class diagram (see Figure 10):

- Apply domain knowledge
  Domain knowledge is applied to remove superfluous flexibility by allowing by design only meaningful object structures. This simplifies the analysability of the design. In this step, inheritance relations are replaced for aggregation relations. The step has a high impact, because it directly involves changes to interfaces. Changes concerns among others the classes CJDMImage, CJDMTemplate, CJDMImageSequence, CJDMFrame and CJDMCarrier. For example, CJDMImage is an aggregate of CJDMFrame instead of a derived class of JDMNode.
- Separate behaviour from the data structure
  Behaviour is relocated to clarify the meaning and simplify the implementations of methods on classes. In our redesign, methods operate only on the attributes and objects that are directly associated to the invoked object. Higher level functions that involve navigation over multiple objects are moved to a facade, which we will discuss later. This operation has a medium impact. It concerns methods from e.g. JDMImageSequence and CJDMNode:
  - navigation from source to target image/frame
  - createnew
  - expand, freeze, updatefromsource, etc.

  For example, the method 'expand' of CJDMImageSequence now searches for the right CJDMTemplate object to expand. Instead we propose to call this method from a facade that implements the navigation service and this method calls the expand method on the appropriate object.
- Apply abstraction and delegation
  Abstraction and delegation simplify design by further reducing the unnecessary coupling between specialised classes. Dependencies between specialised classes are obsolete, because these interfaces can be covered by their base classes. This operation has low impact. It concerns dependencies of specialised classes of CJDMFrame, CJDMCarrier, CJDMDocument, and CJDMNode. For example, the dependency between CJDMSheet and CJDMSheetFrame becomes obsolete if the interaction can be covered by the interface of the respective base classes.

The refactored JDM (see below) is able to express the same functionality as the original, including artefacts like Bitmap documents as a template description. The result of these steps is that the number of classes increased by two. The number of inheritance relations decreased by 4, and as a consequence the dependency relations in JDM become aggregation relations in the redesign. By using proper abstraction, the number of dependencies drops drastically (e.g. the dependency between CJMSheet and CJDMSheetFrame).

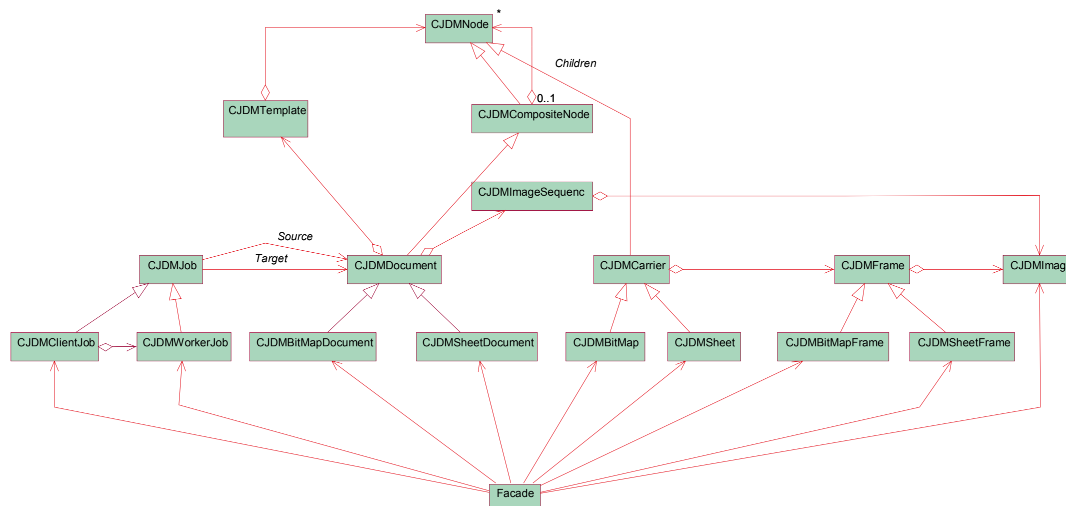

Figure 15 Class Diagram of the refactored JDM

In the diagram above, we introduced a façade [Gamma 95] to provide a simple interface to this complex subsystem. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customisability will need to look beyond the facade. The Splitter typically needs to look behind the facade because this actor is tightly integrated with JDM. Here, we focus on the facade for the actors Client and Worker.

The Client constructs an object using the Client interface. The 'Builder' pattern [Gamma 95] is used to simplify the construction of the job and its source and target document. It isolates code for construction and representation. The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients need not to know anything about the classes that define the product's internal structure. As a consequence such classes do not appear in Builder's interface. The 'Client Interface' (see Figure 16) provides an abstract interface for creating parts of a 'Client Job' object. The specialisations of the 'Client Interface' build the job's internal representation and define the process by which it is assembled. Print Client Interface, Copy Client Interface and Scan Client Interface construct and assemble parts of the job by implementing the Client Interface. They defines and keeps track of the representation they create and provides an interface for retrieving the product (e.g. Client Job).

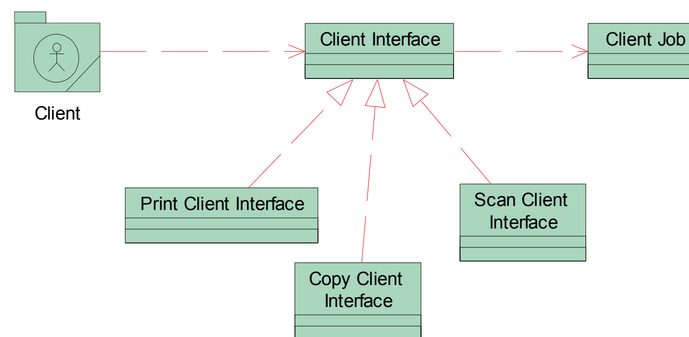

Figure 16 Client interface

The main responsibilities of the Workers concern the transformation of pages to get the job done. In Figure 18, the 'Worker Job' is a specialisation of the abstract class 'Job'. The Worker Jobs are the sub tasks that need to be executed to realise the Client Job. The abstract class 'Job' contains a source and a target document of the type 'Document'. Document provides an interface to the composite that represents the document. Each Document is a 'Shared Resource'. This resource is shared between a producer and a consumer of a page and takes care of the synchronisation. Each worker (Scan, Print, RIP or Image) fulfils both the roles of producer and consumer. The consumer interface provides methods to retrieve the next page from the source document that needs to be transformed. The producer interface enables to retrieve the settings of the page that should result from the transformation and a means to submit the transformed page to the target document.

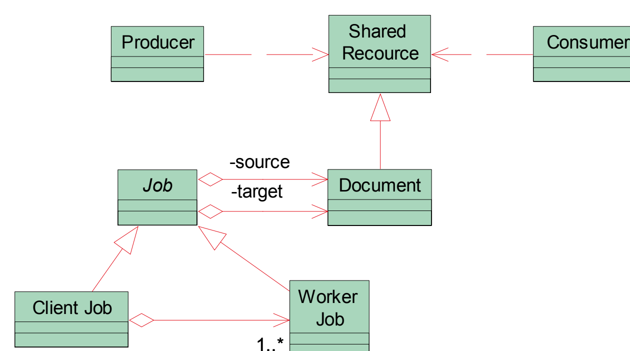

Figure 17 Sharing

The original interface to JDM is a direct mapping of the classes of the model. In total, it exposes 15 classes of its internal model via 15 separate interface classes. Via these interfaces, the complexity is exported to the clients of JDM. The proposed interface to JDM provides a higher level of interaction to the actors of JDM.

Step 8: Dynamic View

Representing and modifying a job structure
The composite pattern is used to express the source and the target document of a job. The implementation of JDM uses (1) a fixed depth of the object structure and (2) the cardinality of relations to express functionality (e.g. duplex is modelled as carrier with two frames). Here, we describe how we make use of this pattern. In essence, we use a flexible object structure, only attributes to express functionality and let the attributes apply to a certain scope (e.g. duplex is an attribute that applies to a sequence of pages). This approach omits superfluous instances of classes and redundant attributes, which is beneficiary to footprint and performance. Attributes are changed more easily then a structure of objects and a functionality that is 'unspecified' can be modelled by the omission of the attribute, which is impossible for a structure.

We base this approach on an analogy with abstract syntax trees that can be described by an attribute grammar. The idea of an attribute grammar is to represent context sensitive information as attributes of the nodes in an abstract syntax tree. An attribute value is defined by a function applied to neighbour node attributes. Such definitions or equations are associated with the production rules.
In the abstract syntax tree, the root and the leafs represent respectively the whole document and the individual pages. All nodes in-between represent the sub documents. Attributes are associated where appropriate to the nodes. All attributes on the path from the root to a leaf apply the page represented by the leaf. In Table 10, an attribute grammar is given for this document representation. The object-oriented notation for attribute grammars suggested by Hedin is used [Hedin 89].

<Node> ::= **Abstract**	
<Document> : <Node> ::= (<attributePart:AttributeList> & <Sub-document>*)	**Loc** rootEnv: Environment; RootEnv:= attributePart.assembleEnv; For all sons(x) in Sub-documentList Son(x).env := env;
<Sub-document> :<Node>:= (<attributePart:AttributeList> & <Sub-document>*)	**Anc** env: Environment; **Loc** subEnv: Environment; subEnv:= attributePart.assembleEnv; For all sons(x) in Sub-document Son(x).env := subEnv;
<Sub-document> :<Node>:= <p:Page>	**Anc** env: Environment; p.env := env;
<Page> : <Node> ::= (<attributePart:AttributeList>)	**Anc** env: Environment; **Loc** pEnv: Environment; PEnv:= attributePart.assembleEnv;

Table 10 Attribute grammar to represent the document structure

The source and target documents are represented by an abstract syntax tree (see above) and describe respectively the start document and the intended document. The tasks or transformations that need to be performed can be derived from the differences in the attribute sets associated to pages that correspond in the source and target document. This difference in attributes follows from the formula below:

$$source.page(x).pEnv - target.page(x).pEnv \cup target.page(x).pEnv - source.page(x).pEnv$$

Support for larger Jobs
The implementation of JDM pre-expands all documents in the job description, which takes time for large jobs. For each page, there exist objects in each document. For a single sided, 5 page copy job, there are currently at least 60 instances from the classes in JDM created, but for a 100 page job, this means 915 instance. In our redesign, we expand the pages of the documents on demand and keep no counter parts, which results for large jobs in a third of the objects used in the original JDM. Below, we explain this mechanism.

Streaming behaviour enables functions like print-while-scan or print-while-rip. This implies that in print-while-scan, the printer starts working before scanner has completed all its work. A scanner

consumes the images of a sheet of paper and produces a bitmap that contains the image. The printer consumes this bitmap and produces a sheet with the image on top. To enable these functions, different workers execute the tasks each in their own thread. The data must be passed from one thread to the other. This hand-off requires a data structure that can hold the chunks of output, pages, from the scanner in the generated order, while the printer retrieves them for processing.

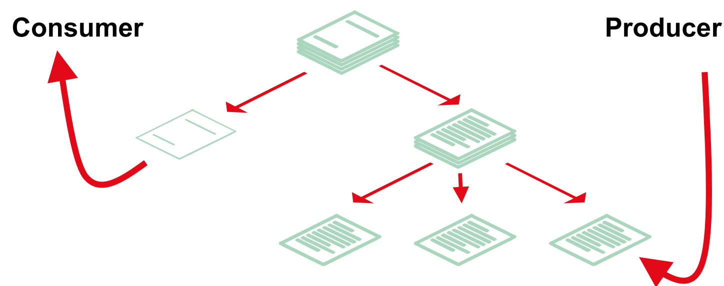

Figure 18 A tree container for Producer/Consumer

The most common data structure for this purpose is the queue. The queue is a first-in-first-out (FIFO) data container. A tree can be considered as a hierarchical queue or can be linearised using in the prefix notation. Data is placed in one end of the data structure and removed from the other in the same order in which it was inserted. Figure 18 shows how this might work for a document that consists of pages.

7 Conclusions and Recommendations

In retrospect, we conclude that JDM was the right place to start to consider improving the maintainability of the controller, because it is crucial in the overall architecture and its potential gain is the largest of all modules on the black board. Moreover, JDM evolved and with its size the complexity of JDM grew linear. In other words if this trend persists, the problems associated to its complexity will increase with each new release of JDM.

The problems with the original design relate to three major topics:
- The mechanisms underlying JDM can not be seen through easily and this complexity is not encapsulated within the module.
- Modifications of the job, especially where it concerns changes to the object structure, are difficult.
- The support for increasingly larger jobs in the controller in the sense of a smaller foot print and better performance (including support for streaming, fine grained notification or improving the locking mechanism).

The friction with respect to changes to the existing job structure can be relieved by (1) specifying all functionality in attributes and not in object structure and (2) on demand expanding of the job structure, because attributes and the unexpanded documents are simpler to change. The support for larger jobs is achieved by (1) on demand expansion of the job structure, (2) cleanup of expansions that are obsolete and (3) specifying only attributes that are set.

Because the number of dependencies has decreased considerably, behaviour and data structure have been separated and functionality is expressed only by means of attributes, our redesign, in general, allows for a much simpler implementation of the features as opposed to the original design of JDM. However, there are a few exceptions, like the predictability of the time to finish a job.

Table 11 presents an overview of important, troublesome features, the chosen solution direction and the improvement (indicated with a minus or plus).

Requirement	Solution in proposed redesign	Result with respect to current design
Predicting e.g. time to finish/number of sheets	On demand calculation, possibly improved with caching	-
Attribute priority default or set	Data structure contains only functionality in attributes and only those that are encountered in the job ticket	+
Changing job structure from mailbox (plexity/covers)	Target document is not expanded yet and feature is only in attribute not in structure	+
Synchronisation	Combination of semaphores and notification mechanism	+/- (only notification, but notification mechanism involves extra programming of queues and handlers)
Persistency	Less attributes; only attributes that are set.	+
Performance	Less objects (approximately third for large jobs), because objects are not pre expanded	+
Streaming	expansion of a page on demand	+
User vs. technical domain	User domain attributes are recorded separately, technical domain attributes are derived from user domain and stored in JDM.	-

Table 11 Redesign compared to current design

We observed that object-oriented metrics, like depth of inheritance and coupling between objects, are helpful to identify the ‘hot spots’ within the software. They are very useful in comparing different entities of the software quantitatively.

In order to circumvent future maintainability problems, which is of course not limited to JDM only, the following 'rules' apply:

1. *Preserve and monitor architectural integrity*
   In practice, carefully chosen and adequate architecture concepts are not always respected due to new and changing requirements during the life cycle of the platform product or time pressure on the project.
2. *Make a good start*
   The most important decisions are made early in project and changes later on have a large impact. The best decisions can be made if the number of uncertainties is low. Uncertainties can relate to changes in process (outsourcing and multi-site development), programming paradigm (object orientation), host environment (MS Windows), technology (e.g. COM) or architecture. Prototyping is an excellent tool to eliminate uncertainties and gain experience.
3. *Change staff gradually*
   With each transition from one phase to the other that involves a large exchange of staff, knowledge is lost. This loss cannot be compensated by documentation alone. However, this loss can be reduced by a period of co-operation between both parties. This period should be long enough to have the key concepts in place and afterwards the departing staff should be available for consultation.
4. *Monitor quality during evolution*
   Software deterioration is a gradual process. Accurate system representations and software metrics, including object-oriented metrics, make trends visible and in time appropriate counter action can be taken. Metrics can form a base for future guidance of non-function requirements, especially maintainability. Even simple metrics, like LOC, Cyclomatic Complexity and the Coupling Between Objects, give clear trends. Automatic tools can facilitate this activity.
5. *Recover shortcuts timely*
   The existing code base usually has a large momentum. Short cuts on top of short cuts make the effort and threshold to recover only higher.

8 Abbreviations

ADF	Automatic Document Feeder
ADL	Architectural Description Language
ATAM	Architecture Trade-off Analysis Method
ARID	Active Reviews for Intermediate Designs
CAM	Capability Model
COM	Component Object Model
IEC	International Electrotechnical Commission
ISO	International Standard Organisation
ITEA	Information Technology for European Advancement
JDM	Job Document Model
KM	Key Mechanisms
LPR	Line Print
MOOSE	Software Engineering Methodologies for Embedded Systems
OP	Object Persistency
PDL	Page Description Language
QSM	Queue and Scheduling Model
REM	Resource Model
RIP	Raster Image Processor
RUP	Rational Unified Process
SAAM	Software Architecture Analysis Method
TAM	Task Model
TU/e	Eindhoven University of Technology

9 References

[Bass 98] Bass, L. Clements, P. Kazman R. "Software Architecture in Practice" Addison-Wesley, 1998

[Bengtsson 99] Bengtsson, P. Bosch, J. "Architecture Level Prediction of Software Maintenance", Proceedings of Third European Conference on Software Maintenance and Reengineering, Amsterdam, The Netherlands, March 1999

[Berard] Berard. E.V. "What Is a Methodology?" http://www.toa.com/

[Booch 94] Booch, G. "Object-Oriented Analysis and Design with Applications", 2nd ed., The Benjamin/Cummings Publishing Company, Inc., 1994

[Bosch] Bosch, J. Bengtsson, P. Smedinga, R. "Assessing Optimal Software Architecture Maintainability"

[Buschmann 96] Buschmann, F. et al "Pattern-Oriented Software Architecture, Volume 1: A System of Patterns" John Wiley & Sons; 1 edition, August 1996)

[Castro 02] Castro, J. Kolp, M. Mylopoulos, J. "Towards Requirements-Driven Information Systems Engineering: The Tropos Project", Information Systems, Elsevier, Amsterdam, The Netherlands, 2002

[Clements 01] Clements, P. Kazman, R. Klein, M. "Evaluating software architectures: methods and case studies" Addison-Wesley, Boston, October 2001

[Dobrica 02] Dobrica, L. Niemelä "A Survey on Software Architecture Analysis Methods", IEE Transactions on Software Engineering, Vol. 28, No. 7, July 2002

[Fowler 98] Fowler, M. "UML Distilled; Applying the Stnadard Object Modeling Language" Addison-Wesley Longman, Inc, January 1998

[Fowler 99] Fowler, M. Beck, K.(Contributor) Brant, J. (Contributor) Opdyke, W. Roberts, D. "Refactoring: Improving the Design of Existing Code" Addison-Wesley Pub Co; June 1999

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

[Gurp 02] Gurp, J. van & Bosch, J. "Design Erosion: Problems & Causes" , Journal of Systems & Software, 61(2), pp. 105-119, Elsevier, March 2002.

[Hedin 89] Hedin, G. "An Object-Oriented Notation for Attribute Grammars", In Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP'89), S. Cook (Ed.), British Informatics Society Ltd., Nottingham, 329-235, 1989.

[Henderson 96] Henderson-Seller, B. "Object-oriented metrics: measures of complexity" Prentice Hall, Inc., 1996

[Heuvel 02] Heuvel, W. J. van de Proper, E. "De pragmatiek van architectuur", informatie, November 2002

[Horowitz 93] Horowitz, E. Sahni, S. Anderson-Freed, S. "Fundamentals of Data Structures in C" W.H. Freeman and Company, 1993

[ISO/IEC 9126] ISO/IEC 9126 "Information technology - Software Product Evaluation - Quality characteristics and guidelines for their use", 1991.

[IEEE 91] IEEE "IEEE Standard Glossary of Software Engineering Terminology" IEEE Software Engineering Standards Collection, Spring 1991 Edition. New York, 1991.

[Kazman 96] Kazman, R. "Tool Support for Architecture Analysis and Design", ACM,. 1996

[Kazman 97] Kazman, R. Carrière, S. J. "Playing Detective: Reconstructing Software Architecture from Available Evidence"

[Kazman 01] Kazman, R. O'Brien, L. Verhoef, C. "Architecture Reconstruction Guidelines", August 2001

[Kim 01] Kim, J. Carlson, C. R. "Design units – a layerd approach for design driven software development", Information and Software Technology, 43, Elsevier, Amsterdam, The Netherlands, August 2001

[Kruchten 95] Kruchten, P. "Architectural Blueprints-The '4+1' View Model of Software Architecture", IEEE Software, Vol. 12, No. 6, November 1995

[Land 02] Land R. "Software Deterioration And Maintainability – A Model Proposal" SERPS, Sweden , October 2002

[Lamsweerde 00] Lamsweerde, A. van "Requirements Engineering in the Year 00: A Research Perspective", Proc. 22nd International Conference on Software Engineering,

Limerick, June 2000, ACM Press

[Lamsweerde 00b] Lamsweerde, A. van “Formal Specification: a Roadmap”, The Future of Software Engineering, A. Finkelstein (ed.), 2000, ACM Press

[Lamsweerde 01] Lamsweerde, A. van “Goal-Oriented Requirements Engineering: A Guided Tour”, Proceedings RE’01, 5th IEEE International Symposium on Requirements Engineering, Toronto, August 2001

[Lehman 85] Lehman, M.M. and Belady, L.A. “Program Evolution Processes of Software Change” Academic Press, London, 1985

[Lehman 01] Lehman, M.M. and Ramil, J.F. “Rules and Tools for Software Evolution Planning and Management” Annals of Software Eng., spec. issue on software management, vol. 11, pp. 15-44, 2001

[Littlefair 01] Littlefair, T. “An Investigation Into The Use Of Software Code Metrics In The Industrial Software Development Environment” Thesis at Edith Cowan University, Mount Lawley Campus., June 2001

[Lorenz 94] Lorenz, M. Kidd, J. “Object-oriented software metrics: a practical guide”, Prentice Hall, Inc., 1994

[Mansurov 02] Mansurov, N. “Using Metrics to Enforce Quality of Managed Architectures” in industrial presentations proc. of int. Conf. Metrics-2002, Ottawa, Canada, 2002

[McCabe 76] McCabe, T.J. “A complexity measure” IEEE Transactions on Software Engineering, December 1976

[Naur 69] Naur, P. Randell, B. “Software Engineering: Report on 1968 Nato Conference”, Scientific Affairs Division NATO, 1969

[O'Brien 01] O’Brien, L. “Architecture Reconstruction to Support a Product Line Effort: Case Study”

[O’Brien 02] O’Brien, L. Stoermer, C. Verhoef, C. “Software Architecture Reconstruction: Practice Needs and Current Approaches”, August 2002 (published October 2002)

[Postema 00] Postema, H. “Architecture Assesment: Approach and Experiences”, LAC, 2000

[Rumbaugh 91] Rumbaugh, J. et al. “Object-Oriented Modeling and Design”, Prentice Hall, Inc., 1991

[Wallace 01] Wallace, S.A. Laird, J. E. Coulter, K. J. “Assessing the Run-Time Performance of Artificial Intelligence Architectures”, Proceedings of the 2000 PerMis Workshop, NIST,Gaitherburg, MD, USA, September 2000

[Yanbing Guo xx] Yanbing Guo, G. Atlee, J. M. Kazman, R. “A Software Architecture Reconstruction Method”

Appendices

Appendix A Metric Suits *36*
Appendix B Models Measures *38*
Appendix C Common and Simple types and interface metric data *41*

Appendix A Metric Suites

This appendix gives a number of common metrics. The first set is from [Lorenz94], the second one is used in the tools CCCC.

Lorenz

CRE	Class reuse	This method considers the classes that are used across projects or are otherwise reused.
FFU	Friend Functions	The metric takes into account all methods that break up the encapsulation of classes.
FOC	Function-oriented code	The metric counts all methods that are not associated to a particular class.
GUS	Global Usage	GUS counts the number of elements that are global to the entire system.
HNL	Hierarchy Nesting Level	The length of the longest path of inheritance ending at the current class.
ISR	Intersubsystem relationships	This metric accounts for the number of intersubsystem messages.
MUI	Multiple Inheritance	This metric counts the number of superclasses from which the class inherits state and behaviour. Not all programming languages support multiple inheritance
NCM	Number of class methods	The figure consists of all static public methods exposed by the class. It takes also the methods from its base classes.
NIM	Number of instance methods	This metric counts all private, protected and public methods of the class in consideration.
NIV	Number of instance variables	The metric counts all variables declared in the scope of the class
NMO	Number of methods overridden	This is the number of instance methods, which signature is available in a method of a base class and thus overrides the behaviour of the base class for the method.
PCM	Percentage of commented methods	This metric gives the percentage of overall commentary included in the methods. The usefulness of the comments is not taken into account; just an annotation is enough.
PCR	Problem reports per class	This metric counts the number problems that are associated to the class.
PIM	Number of pubic instance methods	This number includes all methods (except the con- and destructor) that provide services to its client. It is defined as the PIM of the base class plus the class' total number of public methods minus class' the public class methods minus the number of overridden public methods.
SIX	Specialisation index	This metric is defined as the number of overridden methods multiplied by the hierarchy level divided by the total number of methods. This metric is open for interpretation. All public instance methods and all methods that override a method of a base class are counted.

CCCC

LOC	Lines of Code	This metric counts the lines of non-blank, non-comment source code in a function, module, or project.
MVG	McCabe's Cyclomatic Complexity	A measure of a body of code based on analysis of the cyclomatic complexity of the directed acyclic graph, which represents the flow of control within each function.
COM	Comment Lines	A crude measure, comparable to LOC, of the extent of commenting within a region of code. Not very meaningful in isolation, but sometimes used in ratio with LOC or MVG to ensure that comments are distributed proportionately to the bulk or complexity of a region of code.
L_C	Lines of code per line of comment	Indicates density of comments with respect to textual size of program

M_C	Cyclomatic Complexity per line of comment	Indicates density of comments with respect to logical complexity of program
WMC, WMC1, WMCv	Weighted methods per class	This measure, proposed by Chidamber and Kemerer, is a count of the number of functions defined in a class multiplied by a weighting factor. Two different weighting functions are applied: WMC1 uses the nominal weight of 1 for each function, and hence measures the number of functions, WMCv uses a weighting function which is 1 for functions accessible to other modules, 0 for private functions.
DIT	Depth of inheritance tree	The length of the longest path of inheritance ending at the current class. The deeper the inheritance tree for a class, the harder it may be to predict its behaviour. On the other hand, increasing depth gives the potential of greater reuse by the current class of behaviour defined for ancestor classes.
NOC	Number of children	The number of modules which inherit directly from the current module. Moderate values of this measure indicate scope for reuse, however high values may indicate an inappropriate abstraction in the design.
CBO	Coupling between objects	The number of other modules which are coupled to the current module either as a client or a supplier. Excessive coupling indicates weakness of module encapsulation and may inhibit reuse.
FO, FOv, FOc, FI, FIv, FIc	Fan-out, Fan-in inclusive/visible/concrete	For a given module A, the fan-out is the number of other modules which the module A uses, while the fan-in is the number of other modules which use A. FO Fan out. FOv/FIv is calculated using only relationships in the visible part of the module interface. FOc/FIc is calculated using only those relationships, which imply that changes to the client must be recompiled if the supplier's definition changes.
IF4, IF4v, IF4c	Information Flow inclusive/visible/concrete	A composite measure of structural complexity, calculated as the square of the product of the fan-in and fan-out of a single module. Proposed by Henry and Kafura. IF4v is calculated using only relationships in the visible part of the class interface. IFc is calculated using only those relationships, which imply that changes to the client must be recompiled if the supplier's definition changes.
NOM	Number of modules	Number of non-trivial modules. Non-trivial modules include all classes, and any other module for which member functions are identified.
REJ	Rejected lines	This is a measure of the number of non-blank non-comment lines of code, which was not successfully analysed by the parser. This is more of a validity check on the report generated than a metric of the code submitted: if the amount of code rejected was more than a small fraction (say 10%) of the total code processed, the meaningfulness of the numbers generated by the run must be in doubt.

Appendix B Models Measures

This appendix lists the measurements of the most important, critical classes in the source code of the core modules (including the models TAM, CAM, REM, QSM and JDM, but also KM and OP) of the controller. The tool CCCC[1] provides three types of measurements:

- Procedural measures
  The measures include lines of code, lines of comment and McCabe's Cyclomatic Complexity summed over each class.
- Object-oriented design
  Four of the six metrics proposed by Chidamber and Kemerer in their various papers on 'A metrics suite for object-oriented design' are supported by this tool.
- Structural metrics
  These metrics are based on the relationships of each class with others and includes fan-out (i.e. number of other modules the current module uses), fan-in (number of other modules which use the current module), and the Information Flow measure suggested by Henry and Kafura, which combines these to give a measure of coupling for the module.

We grouped the classes by the modules JDM and QSM. The remainder of the critical classes that originate from different modules are shown in one table. The cells of the tables are coloured yellow (light grey) or red (dark grey), depending of the severity, whenever a class fails a certain pre-set value of a metric. For example the table below marks the module CJTIL for the metric LOC (Line of Code) yellow.

Module Name	LOC	MVG	COM	L_C	M_C	WMC1	WMCv	DIT	NOC	CBO
CJTIL	1957	259	309	6.333	0.838	38	0	0	0	15
CJdmBitmap	223	38	387	0.576	0.098	17	16	7	0	8
CJdmBitmapDocument	269	46	274	0.982	0.168	23	19	7	0	9
CJdmBitmapFrame	270	52	310	0.871	0.168	21	19	7	0	9
CJdmCarrier	175	32	54	3.241	0.593	12	0	6	2	13
CJdmCompositeNode	130	31	205	0.634	0.151	7	7	4	2	6
CJdmDocument	474	101	290	1.634	0.348	22	0	6	2	14
CJdmFrame	210	35	187	1.123	0.187	21	17	6	2	13
CJdmImage	210	35	189	1.111	0.185	19	17	4	0	9
CJdmImageSequence	82	11	77	1.065	0.143	6	0	5	3	7
CJdmIteratorBase	74	5	161	0.460	0.031	14	11	0	1	4
CJdmJob	914	213	498	1.835	0.428	30	0	5	0	12
CJdmNode	1665	469	613	2.716	0.765	65	0	3	3	36
CJdmSheet	246	34	298	0.826	0.114	20	18	7	0	10
CJdmSheetDocument	357	63	330	1.082	0.191	27	22	7	0	11
CJdmSheetFrame	231	38	243	0.951	0.156	19	17	7	0	7
CJdmStack	92	10	95	0.968	0.105	9	9	0	0	3
CJdmTemplate	838	190	687	1.220	0.277	38	0	4	0	11

Table 12 Procedural and OOD Measures of JDM

[1] CCCC - C and C++ Code Counter – is a free software tool for measurement of source code related metrics by Tim Littlefair. The CCCC tool was developed as a testing ground for a number of ideas related to software metrics in a MSc project. The research project is described at http://www.fste.ac.cowan.edu.au/~tlittlef

	Fan-out			Fan-in			IF4		
Module Name	vis	con	inc	vis	con	incl	vis	con	inc
CJTIL	0	0	0	15	12	15	0	0	0
CJdmBitmap	1	0	1	6	3	7	36	0	49
CJdmBitmapDocument	0	0	0	9	3	9	0	0	0
CJdmBitmapFrame	0	0	0	8	4	9	0	0	0
CJdmCarrier	5	2	5	8	4	8	1600	64	1600
CJdmCompositeNode	2	2	2	3	2	4	36	16	64
CJdmDocument	3	2	3	11	5	11	1089	100	1089
CJdmFrame	5	2	5	8	3	8	1600	36	1600
CJdmImage	3	0	3	6	3	6	324	0	324
CJdmImageSequence	3	3	3	4	2	4	144	36	144
CJdmIteratorBase	1	1	1	3	2	3	9	4	9
CJdmJob	0	0	0	12	5	12	0	0	0
CJdmNode	18	3	18	18	8	18	104976	576	104976
CJdmSheet	2	0	2	8	4	8	256	0	256
CJdmSheetDocument	0	0	0	11	5	11	0	0	0
CJdmSheetFrame	0	0	0	7	3	7	0	0	0
CJdmStack	0	0	1	2	1	2	0	0	4
CJdmTemplate	1	0	1	10	3	10	100	0	100

Table 13 Structural Measures of JDM

Module Name	LOC	MVG	COM	L_C	M_C	WMC1	WMCv	DIT	NOC	CBO
QsmBasicJob	722	116	690	1.046	0.168	33	32	3	0	17
QsmCompJob	1010	195	608	1.661	0.321	36	31	3	0	18
QsmJobControl	341	55	388	0.879	0.142	22	22	3	0	12
QsmMailboxListOfRequests	463	114	430	1.077	0.265	15	8	3	0	9
QsmMailboxRequest	230	45	276	0.833	0.163	12	11	3	0	7
QsmRequest	1084	212	757	1.432	0.280	38	29	3	0	15
QsmWorker	363	47	479	0.758	0.098	26	26	3	0	11
QsmWorkerQOfBasicJobs	399	66	399	1.000	0.165	23	16	3	0	6

Table 14 Procedural and OOD Measures of QSM

	Fan-out			Fan-in			IF4		
Module Name	**vis**	**con**	**inc**	**vis**	**con**	**incl**	**vis**	**con**	**inc**
QsmBasicJob	0	0	0	17	14	17	0	0	0
QsmCompJob	0	0	0	17	15	18	0	0	0
QsmJobControl	0	0	0	11	10	12	0	0	0
QsmMailboxListOfRequests	0	0	0	7	6	9	0	0	0
QsmMailboxRequest	0	0	0	6	5	7	0	0	0
QsmRequest	0	0	0	12	12	15	0	0	0
QsmWorker	0	0	0	10	11	11	0	0	0
QsmWorkerQOfBasicJobs	0	0	0	4	5	6	0	0	0

Table 15 Structural Measures of QSM

Module Name	LOC	MVG	COM	L_C	M_C	WMC1	WMCv	DIT	NOC	CBO
BimBitmapProxy	617	96	115	5.365	0.835	33	17	3	0	11
CRCList	67	6	2	33.500	3.000	10	9	0	0	11
CRCMap	95	10	9	10.556	1.111	11	8	0	0	5
Capabilities	67	4	171	0.392	------	6	6	3	0	8
RemObjectFactory	101	9	9	11.222	1.000	4	4	0	0	1
RemResourceComponent	868	124	130	6.677	0.954	40	31	3	1	16
RemStrings	174	41	9	19.333	4.556	10	8	0	0	6
RemSystem	223	27	31	7.194	0.871	33	29	4	0	11
RemXMLDoc	268	46	58	4.621	0.793	6	3	0	0	7
RemXMLResource	83	2	10	8.300	------	4	3	0	0	7
TamContainer	227	28	62	3.661	0.452	10	7	3	2	9
TamInteraction	125	8	40	3.125	0.200	6	6	4	0	4
TamTask	537	54	235	2.285	0.230	24	17	4	0	6
anonymous	3731	160	4820	0.774	0.033	146	31	0	0	0

Table 16 Procedural and OOD Measures of the Remaining models

	Fan-out			Fan-in			IF4		
Module Name	**vis**	**con**	**inc**	**vis**	**con**	**incl**	**vis**	**con**	**inc**
BimBitmapProxy	0	0	0	10	8	11	0	0	0
CRCList	1	7	8	2	2	3	4	196	576
Capabilities	0	0	0	7	4	8	0	0	0
RemObjectFactory	0	0	0	1	1	1	0	0	0
RemResourceComponent	1	1	1	13	12	15	169	144	225
RemStrings	0	0	0	6	5	6	0	0	0
RemSystem	0	0	0	10	7	11	0	0	0
RemXMLDoc	2	0	2	3	4	5	36	0	100
RemXMLResource	1	0	1	5	3	6	25	0	36
TamContainer	2	2	2	7	4	7	196	64	196
TamInteraction	0	0	0	3	4	4	0	0	0
TamTask	0	0	0	4	5	6	0	0	0
anonymous	0	0	0	0	0	0	0	0	0

Table 17 Structural Measures of the Remaining Models

Appendix C Common and Simple types and interface metric data

This appendix complements the list of measurements of the critical classes that the tool CCCC identified in the source code of the controller. We grouped the classes by in common, simple types and interfaces. The cells of the tables again are coloured yellow (light grey) or red (dark grey), depending of the severity, whenever a class fails a certain pre-set value of a metric.

Module Name	LOC	MVG	COM	L_C	M_C	WMC1	WMCv	DIT	NOC	CBO
CException	22	2	105	0.210	------	4	4	0	7	9
COp2Storage	880	135	166	5.301	0.813	28	0	0	0	2
CTriggerGenerator	89	7	223	0.399	0.031	11	6	1	0	8
CoStorage	156	20	35	4.457	0.571	13	11	1	0	5
CoStream	89	7	88	1.011	0.080	14	14	1	0	2
Dispatcher	93	10	212	0.439	0.047	8	5	0	0	8
DispatchingQueue	137	13	91	1.505	0.143	9	6	0	0	9
IBBMonitor	44	1	54	0.815	------	19	11	0	0	8
KMArray	123	13	41	3.000	0.317	16	12	0	0	4
KMObject	71	8	95	0.747	0.084	11	9	2	22	26
PersistentNWIC	89	9	5	17.800	1.800	14	14	0	1	3
PersistentWIC	129	10	97	1.330	0.103	19	17	0	1	4
PersistentWICW	53	4	3	17.667	------	4	4	0	0	1
RefCountPtr	31	3	4	7.750	------	8	8	2	0	3
SmartPtr	213	27	19	11.211	1.421	24	20	1	1	4
Subscriber	176	30	149	1.181	0.201	11	0	0	0	7

Table 18 Procedural and OOD Measures of Common

	Fan-out			Fan-in			IF4		
Module Name	vis	con	inc	vis	con	incl	vis	con	inc
CException	7	7	7	2	2	2	196	196	196
COp2Storage	0	0	0	2	1	2	0	0	0
CTriggerGenerator	0	0	0	5	7	8	0	0	0
CoStorage	0	0	0	2	4	5	0	0	0
CoStream	0	0	0	1	1	2	0	0	0
Dispatcher	3	0	3	3	4	5	81	0	225
DispatchingQueue	3	0	3	4	6	6	144	0	324
IBBMonitor	0	0	0	3	5	8	0	0	0
KMArray	0	0	0	3	2	4	0	0	0
KMObject	23	22	23	3	2	3	4761	1936	4761
PersistentNWIC	1	1	1	2	1	2	4	1	4
PersistentWIC	1	1	1	2	1	3	4	1	9
PersistentWICW	0	0	0	1	0	1	0	0	0
RefCountPtr	0	0	0	3	1	3	0	0	0
SmartPtr	1	1	1	3	2	3	9	4	9
Subscriber	0	0	0	7	4	7	0	0	0

Table 19 Structural Measures of Common

Module Name	LOC	MVG	COM	L_C	M_C	WMC1	WMCv	DIT	NOC	CBO
CRITICAL_SECTION	0	0	0	------	------	0	0	0	0	15
HANDLE	0	0	0	------	------	0	0	0	0	7
IBB_INT32	0	0	0	------	------	0	0	0	0	14
IbbAttributeIdEnum	0	0	0	------	------	0	0	0	0	13
IbbValueStruct	0	0	0	------	------	0	0	0	0	14
NotificationPtr	0	0	0	------	------	0	0	0	0	9
RC_ERROR	0	0	0	------	------	0	0	0	0	7
REFIID	0	0	0	------	------	0	0	0	0	27
bool	0	0	0	------	------	0	0	0	0	27

Table 20 Procedural and OOD Measures of Simple Types

	Fan-out			Fan-in			IF4		
Module Name	**vis**	**con**	**inc**	**vis**	**con**	**incl**	**vis**	**con**	**inc**
CRITICAL_SECTION	2	15	15	0	0	0	0	0	0
HANDLE	2	7	7	0	0	0	0	0	0
IBB_INT32	13	14	14	0	0	0	0	0	0
IbbAttributeIdEnum	13	13	13	0	0	0	0	0	0
IbbValueStruct	14	10	14	0	0	0	0	0	0
NotificationPtr	9	8	9	0	0	0	0	0	0
RC_ERROR	7	0	7	0	0	0	0	0	0
REFIID	27	27	27	0	0	0	0	0	0
bool	23	20	27	0	0	0	0	0	0

Table 21 Structural Measures of Simple Types

Module Name	LOC	MVG	COM	L_C	M_C	WMC1	WMCv	DIT	NOC	CBO
IBimBitmapProxy	20	0	101	0.198	------	15	15	0	1	6
ICoCam	46	0	3	15.333	------	4	0	0	0	6
ICoResourceComponent	111	3	6	18.500	------	24	0	0	0	7
ICoTamTask	45	0	3	15.000	------	15	0	0	0	2
IJTIL	23	0	4	5.750	------	4	0	0	0	7
IOp2Storage	11	0	14	------	------	7	7	0	1	25
IOp2Stream	9	0	23	------	------	4	4	0	1	26
ITamTask	22	0	80	0.275	------	17	17	0	1	3

Table 22 Procedural and OOD Measures of Interface Classes

	Fan-out			Fan-in			IF4		
Module Name	**vis**	**con**	**inc**	**vis**	**con**	**incl**	**vis**	**con**	**inc**
IBimBitmapProxy	1	1	1	5	1	5	25	1	25
ICoCam	0	0	0	6	3	6	0	0	0
ICoResourceComponent	0	0	0	7	4	7	0	0	0
ICoTamTask	0	0	0	2	1	2	0	0	0
IJTIL	0	0	0	7	4	7	0	0	0
IOp2Storage	24	1	24	1	0	1	576	0	576
IOp2Stream	26	1	26	0	0	0	0	0	0
ITamTask	1	1	1	2	1	2	4	1	4

Table 23 Structural Measures of Interface Classes