FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

iFAct Recoding

Luís Roma Pires

Report of Project/Dissertation
Master in Informatics and Computing Engineering

Supervisor: João Manuel Paiva Cardoso (Associate Professor)

29th June, 2009

iFAct Recoding

Luís Roma Pires

Report of Project/Dissertation
Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: Pedro Alexandre Guimarães Lobo Ferreira do Souto (Assistant Professor)

External Examiner: João Alexandre Baptista Vieira Saraiva (Assistant Professor , Universidade do Minho)
Internal Examiner: João Manuel de Paiva Cardoso (Associate Professor)

16st July, 2009

Abstract

There are billion of lines of code in existent software and about 80% of them are unstructered, patched or badly documented [EFK+03]. SOFTWARE MAINTENANCE is the most costly and enduring phase in the Software Development life cycle [SB07].

Being the last step, Maintenance efforts levels depend mostly on the quality of the previous phases of the whole Development process. However, this indication is useless for existing software. Insufficient documentation, poor structuring, and unintelligible implementation, may impede the success in software development, and therefore reduce the maintainability of a product. Maintenance in adverse conditions is therefore inevitable.

Taking the opportunity to facilitate Maintenance of a software product that presents any or some of the afore mentioned problems is a responsible measure and will create conditions to greatly reduced the effort invested in the process of correcting or improving the existent software, allowing a more tractable maintenance.

A project in Maintenance may need (re)creation of requirements, architecture and design documentation through reverse engineering processes, as well as redefinition all of these aspects.

Software recoding is the last step of the overhauling of Software, making the code conformant with the product design and also making it more readable, consistent and structured.

This report presents the work done during a first step on the recoding of iFAct, a project in maintenance with a maintainability below the desirable level. It describes all the application of the ideas referred above on the steps taken on the recoding, from understanding the software business model to redefining the architecture for the existing applications and finally to the creation of a prototype design based on the defined architecture and its application to a proof of concept through the refactoring of the existent code.

Resumo

Existem biliões de linhas de código em uso permanente e cerca de 80% desse código não é estruturado, está remendado ou mal documentado [EFK+03]. MANUTENÇÃO DE SOFTWARE é a fase do ciclo de vida de desenvolvimento de software mais cara e duradoura [SB07].

Sendo a última etapa do ciclo de vida, os níveis de esforço despendido em Manutenção dependem principalmente da qualidade do trabalho realizado nas fases anteriores de todo o processo de desenvolvimento. No entanto, essa indicação é inútil para o software já existente. Documentação insuficiente, má estruturação e implementação ininteligível, podem impedir o sucesso no desenvolvimento de software e, consequentemente, reduzir a sustentabilidade de um produto. Manutenção em condições adversas é, portanto, inevitável.

Aproveitar a oportunidade de facilitar a manutenção de um produto de software que apresenta algum ou alguns dos problemas acima mencionados é uma medida responsável e cria condições para reduzir bastante o esforço investido no processo de corrigir ou melhorar o software existente, o que permite uma manutenção mais fácil.

Um projecto de Manutenção pode exigir (re)criação de requisitos, arquitetura e design, documentação, tudo isto através de processos de engenharia reversa, bem como a redefinição de todos estes aspectos.

Recodificação é a última etapa do melhoramento de software, tornando o código concordante com a concepção do produto, para além de também o tornar mais legível, coerente e estruturado.

Este relatório apresenta o trabalho realizado no âmbito da recodificação do iFAct, um projeto de manutenção com uma sustentabilidade abaixo do nível desejável. É descrita a aplicação de todas as ideias acima referidas sobre as medidas tomadas para a recodificação, desde a compreensão do modelo de negócio do software até à redefinição da arquitetura para as aplicações existentes e, finalmente, para a criação de um design baseado na arquitectura e na sua aplicação através de uma prova de conceito recorrendo ao Refactoring do código existente.

Acknowledgements

I'd like to thank my parents, Fernando and Graça, and my relatives for their never ending patience for my bad temper after long hours of work and for their comprehension of what I have to do because I just have to do it.
To my course colleagues and friends throughout all these years, for the great environment we created amongst each other that helps us to acquire knowledge on all kind of areas in Informatics and also to keep up our spirits.
To my team, my friends, and coworkers at Critical Software for giving me a cheerful working place where one can never feel bad and for the openness shown from day one.
To all the teachers I've had in my life for all those inevitable barriers that made me grow up.
To "my" groups: GdF, IEPC and EPC, TdT, Ni, EVC, MCV. To my *consigliere* for being more than advisors and my eyes at FEUP but also good friends that always make me feel great and that drive me towards history.

To *Engenharia* for making me who I am today.

And last, but certainly first in my heart, to Bárbara.

Luís Roma Pires

"People don't think."

Anonymous

Contents

List of Figures

LIST OF FIGURES

List of Tables

Acronyms

CRUD Create Retrieve Update and Delete. 35

GPS Global Positioning System. 2

ICs Integrated Circuits. 1

LDAP Lightweight Directory Access Protocol. 8

LEDs Light-Emitting Diodes. 2

MVC Model-View-Controller. 23, 24

OOP Object Oriented Paradigm. 7, 16, 18, 20

PHP PHP: Hypertext Processor. 23

PV Passive View. 23

SC Supervising Controller. 23, 24

Chapter 1

Introduction

This thesis is the result of the knowledge and experience acquired during the execution of a Master's project named *iFAct Recoding* developed for sixteen weeks at Critical Software [SA09], Oporto offices.

1.1 Context

Critical Software is a provider of solutions, services and technologies for mission and business critical information systems on such different markets as Government, Defense, Finance, Energy, Manufacturing, Telecommunications, Aerospace industries. Founded in 1998, Critical Software currently has offices in Coimbra, Lisbon and Oporto (Portugal), San Jose, California (USA), Southampton (UK) and Bucharest (Romania) and has been growing continuously, presenting in 2007 a turnover of €13.7M, a 60% growth on the previous year, 70% of which accounted from international trades [SA09]. iFAct is a software application used in the reliability and quality control cycle of the production of semiconductors and chips at Infineon Technologies. iFAct was designed in order to assist the automation of analysis requests and reporting procedures in Infineon's failure analysis laboratories iFAct also has other non-core feature such as management and tracking of job analysis and their operations, costs, etc.

iFAct was brought to Critical Software by Infineon Technologies [AG09a] exclusively for maintenance. It was originally developed by another software company.

Infineon Techonologies produces and markets a broad variety of electronic components for industrial and consumer goods. The company's portfolio includes:

- Integrated Circuits (ICs) (automotive, mobile phones, power management)
- microcontrollers

- Light-Emitting Diodes (LEDs)
- sensors (temperature, magnetic, pressure)
- transceivers (Bluetooth, Wireless, ...)

Infineon's products are applied in a large variety of areas such as automotive, mobile phones and Global Positioning System (GPS), lightning, consumer goods, data processing, medical, power control, transportation and renewable energies [AG09b].

Semiconductors are materials which have a conductivity between conductors (general metals) and nonconductors or insulators (such as ceramics) [Ass09a]. Semiconductors are frequently made with Silicon, although a myriad of other elements are used in its production, like Gallium Arsenide, Silicon Carbide or Germanium [Ass09b].

The role played by this product in the fabrication of electronic devices makes them very important in modern lifestyle and to the overall economy, as can be seen with the figure of $USD 249 thousand millions of sales in 2008 [Ass09c]. Some of the main players in this industry are Taiwan Semiconductor Manufacturing Company, United Microelectronics Corporation, Intel, Toshiba, NEC, Sony, Texas Instruments, ST Microelectronics and NXP Semiconductors.

1.2 Motivation and Goals

This project's main goal was the development of an architecture that worked as a basis for the recoding of iFAct that allowed to reduce the current maintenance efforts of the product as it is. The requirements involved with this goal where:

- Identification of common code used on iFAct's applications
- Creation of a new layer with common functionalities while optimizing it's flow.

With the advent of software the need for its modification, correction, adaption and improvement was immediately born. As stated in Lehman's laws of program evolution, software systems are condemned to change over time or become progressively less useful [Leh80].

What obliges this constant evolution is both the need to make an application better (by developing new features, making it perform better in terms of time and resources consumption and adapting it to the environment that surrounds it) and the urge to make the application perform more accordingly to what is desired. However maintenance is not the most expensive phase in software's life cycle because of its obligatory existence, but because of mistakes made in the previous stages of software development. iFAct suffered with the defects on the definition of its design. The existing arquitecture had flaws that increased the maintenance effort necessary for both corrective and evolutive interventions.

1.3 Project

This project's dealt with iFAct: a dual application software product that provides automation support for the analysis requests and reporting procedures in Infineon's Failure Analysis labs.

iFAct consists on a desktop application for Infineon's laboratories employees and a web application for the analysis' submitters to have access to the results of their requests.

This project consisted in planing the recoding of these two applications in order to create a robust, easily maintainable and well documented architecture and a concrete design that can work as a guidebook for the complete recoding of the applications.

Work on the project, in order to fulfill its goal, included:

- Analysis of the business model of failure analysis of Infineon's products;
- Study of the current application's architecure;
- Study of the state of the art in software maintenance and software refactoring;
- Selection and adaptation of an architectural pattern to the project needs (both the application and its recoding process);
- Definition of a roadmap for the refactoring to the defined architecture;
- Development of a Proof of Concept in one of the areas of the application's scope.

1.4 Thesis Structure

This report is organized in the following chapters:

In chapter 2 - *iFAct, a project in maintenance* — An in depth description of iFAct, the application on which the work reported in this thesis was applied, covering it's features and technologies and the reasons for performing this project.

In chapter 3 - *State of the Art* — A walkthrough on the existent methods of Refactoring and Design Patterns applied to Software Maintenance followed by an analysis of the technologies available for the described methods.

In chapter 4 - *Solution Specification* — A description of the proposed solution and the methods followed to reach for the iFAct Recoding, accompanied by a clear explanation of each of the decisions made throughout the project.

In chapter 5 - *Proof of Concept* — A description of the application of the proposed solution in a proof of concept: the implementation of a specific group of features of iFAct.

In chapter 6 - *Conclusions* — A revision of the project, drawing it's achievements, the necessary conclusions and the impact of the solution on the software maintenance process, namely concerning future work that might be done.

Chapter 2

iFAct, a project in maintenance

2.1 Description

iFAct was a maintenance project that suffers from misfit software development. The available documentation was very scarce with no requirements and no functional specification available, only one architecture specification and three small design specification documents, all of which depicting the application at a very high level, with almost no description of the systems interface. The metrics in Table 2.1 help understanding iFAct's complexity.

Metric	iFAct Lab	iFAct Web
Number of Classes	293	212
Methods	4325	2450
Lines of Code	147.363	41.345
Max Complexity	112	42
Average Complexity	3,29	2,32
Maitainability Level	12	14

Table 2.1: Code Metrics

The metrics presented in (calculated with Source Monitor [Sof09]) show iFAct's very respectable dimension. The Complexity value is measured according to a Steve McConnell metric, that accounts the number of execution paths through a function or method, originated by conditional statements and loops [McC93]. The higher the value the more complex the code is, and therefore its understandability is lower.

The Max Complexity values presented are the maximum absolute value of the number of execution paths defined in any function of the iFAct applications.

The Average Complexity value is the average complexity of all the methods, and it's low value compared with the max value can be justified by the high number of methods that have very low usefulness.

The Maintainability Level (explained further ahead in the section 3.2) is nearly

Despite the code's size and complexity the only reliable software representation of iFAct was the code itself, which was messy, with weak structure and only with comments to explain methods.

2.1.1 Software's Goals

iFAct is part of Infineon's software collection used for the reliability and quality control of the production of semiconductors. iFAct is designed for the company's failure analysis laboratories to assist the automation of analysis requests and reporting procedures as well as management and tracking of job analysis and their operations. Infineon's failure analysis laboratories receive jobs from Infineon's reliability process. The engineer labs use iFAct to get data for the tests they run on Infineon's products, to report back the results of the tests, and also to measure their own working effort.

iFAct Lab is responsible for the following tasks in the scope of the failure analysis laboratories:

- Standardization and classification of the analysis results
- Automatic generation of reports
- New job searching
- Handling of images generated by lab equipment
- Accounting of working time
- Tracking of analysis Jobs and respective results
- Statistical evaluations (monthly reports, accounting...)

iFAct Web can be seen as a subset of the Lab application and provides to clients the following features:

- Access to analysis results;
- Job status tracking;
- Job submission.

2.2 Applications Architecture

High Level Architecure iFAct is divided into two applications: *Lab* and *Web* with the first being used by laboratory engineers and managers (in the performance of their day-to-day tasks at the laboratories) and the former used by analysis submitters to have access to the results of theirs requests.

Figure 2.1 shows the basic interactions between the interface modules of iFAct (Lab and Web), the Image Server (that keeps image created by the laboratories machines), the File Server (that keeps the reports created in the laboratories with iFAct Lab), and RSLSecurity (the module responsible for iFAct's security concept).

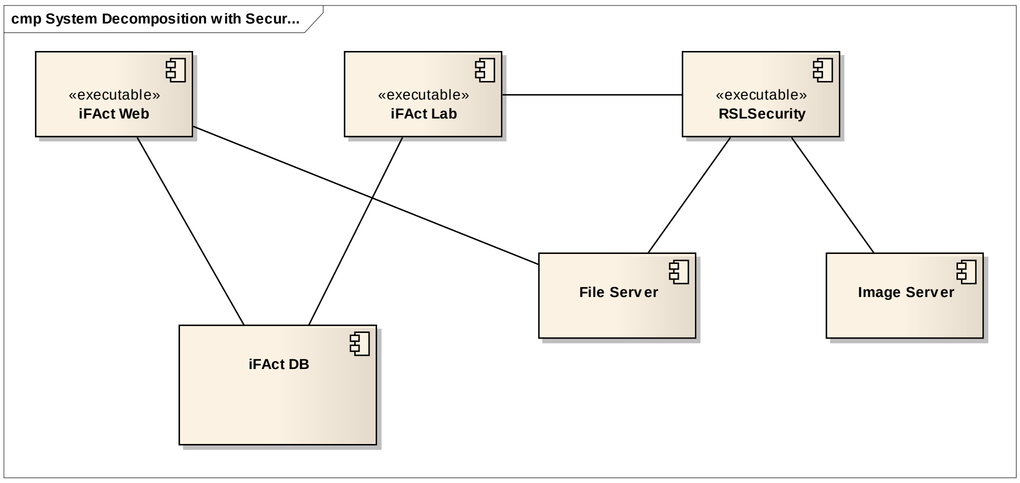


Figure 2.1: High-level system decomposition (with security)

iFAct's high level architecture is, in our opinion, a clean and simple architecture that serves its basic needs and integrates well in the context it works in (see subsection 2.2.1 Environment Integration).

Design iFAct's design was its worse flaw. A big number of anti-patterns were registered and easily identifiable in its structure. Code that was duplicated, unclear and complicated resulted in the most common design problems [Ker04] and all these three could be found in iFAct.

Developed in C#, an Object Oriented Paradigm (OOP) language it took almost no advantage from its potential.

There was no architectural pattern used to separate basic User Interface functions from work flow control and from database activities. There was an entanglement of methods responsible for Business Logic and Interface and also for the Data Layer. where it should be found an N-Layered architecture we found *Windows Form* Classes that beside creating the user interface also handled all the business logic and most of the data access flow.

Because Business Logic was not objectively defined, there was no clear data definition to work with on the software's work flow. A lot of Business concepts were mixed and sometimes repeated up to three times.

There are too many methods that are too long and also a lot of methods that are short and inexpressive. Some of the before mentioned long methods have a lot of code repetitions amongst them that clearly should be isolated in methods.

All of these code smells (also known as anti-patterns) make iFAct's code very difficult to read and understand and therefore the buy-in period in this project was very long.

2.2.1 Environment Integration

iFAct integrates itself within a variety of Infineon's systems as depicted in Figure 2.2. iFACt's interaction with the systems consists in:

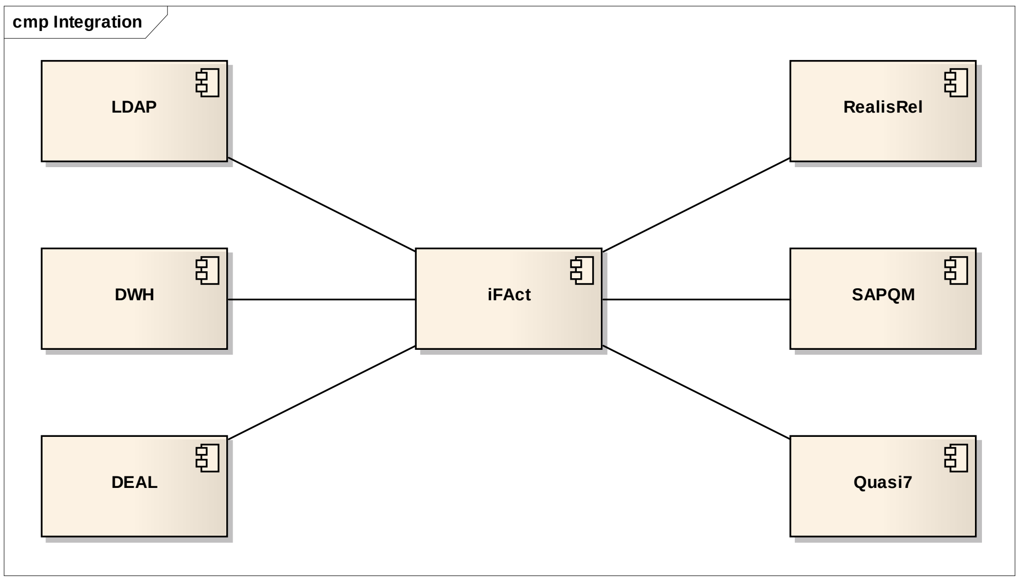


Figure 2.2: Application Integration Diagram

- Receive job from RealisRel, SAP Quality Management and Quasi7
- Get product information from DEAL and DWH databases
- Get Infineon employees information from Lightweight Directory Access Protocol (LDAP)

It is important to note that the interface between iFAct and RealisRel, SAP Quality Management and Quasi7 is the iFAct database. All three systems access iFAct's Database to acquire data and to insert new jobs directly.

2.2.2 Business Description

The Job *submission* process can be started manually by some actor, either by the direct introduction by an iFAct user, through Mandatory Manager quick job submission, or started by RealisRel, Quasi7 or SAP Quality Management. Figure 2.3 shows the life cycle of a job in iFAct.

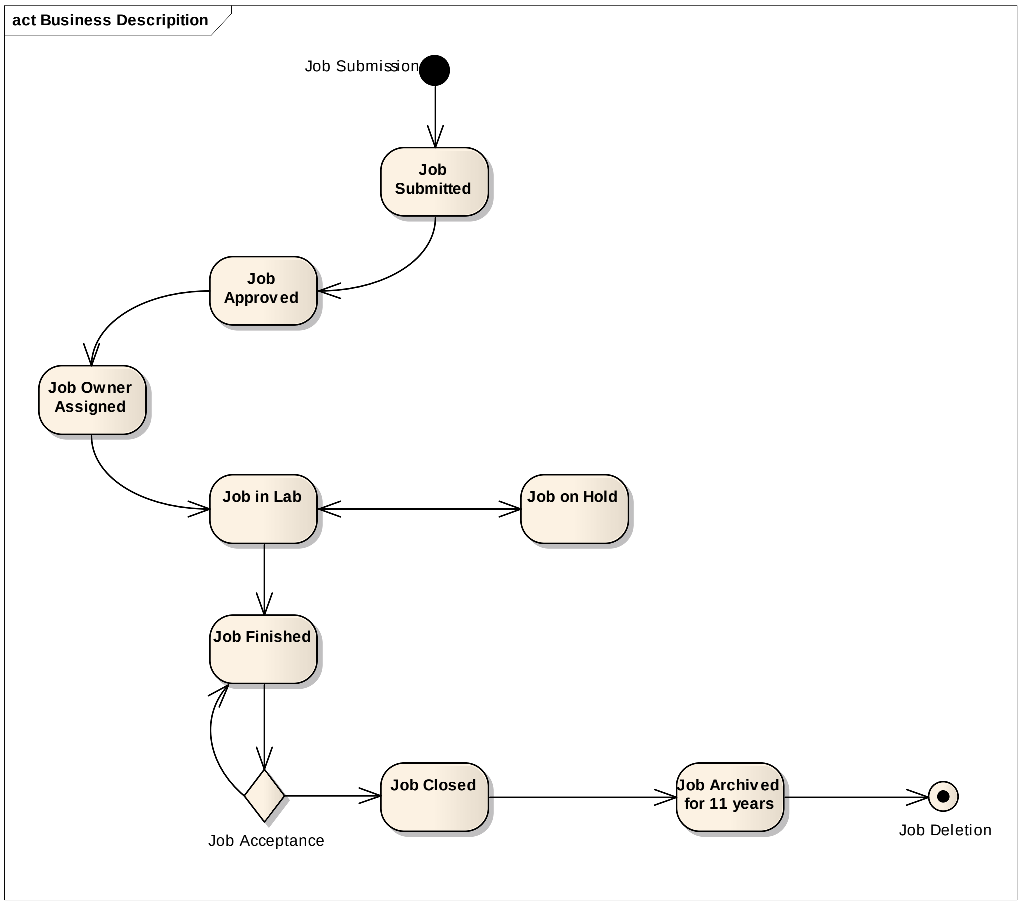


Figure 2.3: Main Business Description

After the submission process the job waits for an *approval* from a laboratory manager. After the job's approval, it is *assigned* to an owner, who is responsible for performing it. When the job has an owner it is said to be *in lab*, meaning the job is being processed. If the job must be halted (e.g., because of working schedules), the job is set to be *on hold*, waiting to be resumed later on.

Eventually the job is *finished* and awaits for its requester to evaluate the conclusion of the job. It will either be rejected and therefore sent to lab or it will be accept and *closed*.

After being closed all jobs are *archived* for at least 11 years for registry.

2.3 Technology Base

iFAct was developed by another software company and three years ago it was brought by Infineon Technologies to Critical Software for maintenance.

2.3.1 Development Languages

iFAct was originally developed using Microsoft® .Net Framework 1.1, iFAct Lab with C# and iFAct Web ASP.Net Framework supported by C#. Later, iFAct was ported to .Net Framework 2.0. Albeit this development upgrade, iFAct Lab versions are released for both frameworks because not all Infineon sites have the 2.0 framework available.

iFAct Lab application uses four commercial third parties (Office Primary Interop Assemblies, Infragistics NetAdvantage, SyncFusion and Infragistics DataWidgets) for the creation of parts of the user interface (namely some data grids and tree views) and an API for the creation of Microsoft®Word®, Excel®and PowerPoint®documents and Outlook®messages.

To avoid the usage of third parties is one of the goals of the maintenance project.

2.3.2 Database Management System

iFAct's is supported by an Oracle Database that started with version 8, when the application was created, and has been updated to versions 9i and to 10g, currently in use.

2.4 Maintenance effort statistics

As a reference, in the last nine months, the effort in the iFAct project was 140% of the initilly expected [1]. Also in Table 2.2 it is shown the effort division amongs the maintenance tasks.

Corrective Maintenance	45%
Preventive Maintenance	31%
Perfective or Adaptive Maintenance	19%
Training and other costs	5%

Table 2.2: Maintenance effort statistics

Although without a comparison to other projects available, it is believed the values shown in are representative of the low level of maintainability of the project, represented by the very high percentage of effort applied in correction of errors and trying to make the project more maintainable.

[1] Internal Comunication, 2009

2.5 Reasons for Recoding

Being responsible for the maintenance of iFAct's code, it was understood by Critical Software that it was the right time to make an investment in order to reduce the effort applied in this maintenance project.

The nearly inexistent project documentation (the only existent documentation is the result of a reverse engineering effort at the time the project was adjudicated to Critical Software) and the very unstructured and design-less code are a big impediment to the work of the project's personnel, especially for newcomers. It was also taken in account that iFAct has been in maintenance for a long time with the natural effects on maintainability depicted in Figure 2.4.

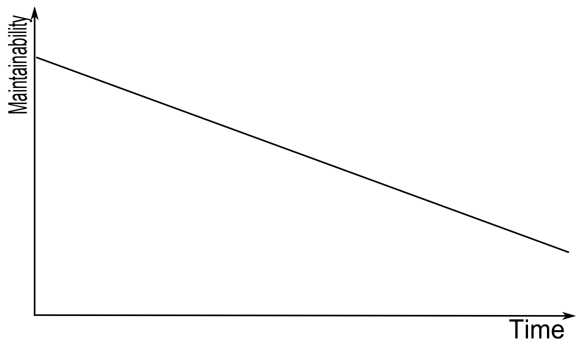


Figure 2.4: Evolution of Maintainability over Time

The existence of some code replications was also an important factor in making the decision of recoding, as it might create incoherences besides the need for a bigger effort in maintenance: the correction of a defect or improvement of a functionality might require as much as twice or more times the work if there are two or more repeated code sections or different code that implements the same functionalities in different parts of the application.

Chapter 3

State of the Art

Given the conditions specified before, accounted in section 2.5 Reasons for Recoding, it was decided to study more about Maintenance, Design Patterns and Refactoring. This study was made in order to make the most of code reusage to achieve a good design that respects iFAct's needs, especially in terms of maintainability.

3.1 Maintenance

Maintenance, being part of the last phase of software development life cycle, is greatly dependent on the quality of the work done on the previous phases: Requirements Engineering, Design, Implementation, Testing, Deployment [Som07]. The secret for easily maintained software is developing a good output in each of these stages allowing for a better knowledge of what software does, how it interacts with its environment and what parts should be tweaked to create the necessary changes. This knowledge is called *Program Understanding* [EFK+03]. Mistakes made in previous phases increase the chance of errors in the later stages.

The IEEE defines Maintenance as follows [oEE98]:

> "the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

A carefully taken software development process, with a view to the software's future and its evolution needs, reduces the effort of all the tasks encompassed in the maintenance. Good documentation (both of the code and the software's design) and good programming practices (especially the usage of error logging) reduce the effort on code understanding and error tracing, conducting the efforts to the software evolution in itself.

On the other hand, software developed with disregard to the need of work to be done in the future brings very hard work to maintenance tasks, diverting attention to understanding what code does, debugging and sometimes causing big time losses when just a small change is needed.

The level of Program Understanding is therefore a key element in software maintenance for its successful correction and evolution as desired or necessary [EFK+03].

3.1.1 Forms of Maintenance

Four different types of software maintenance can be identified [oEE98].

- Corrective Maintenance
- Adaptive Maintenance
- Perfective Maintenance
- Preventive Maintenance

From these four types, corrective maintenance can be considered the '*traditional*' software maintenance while the other are seen as *software evolution* activities.

Corrective Maintenance is a reactive activity, normally triggered by a *Maintenance Request* that addresses discovered faults or defects in a software product [EFK+03]. This is the most common and easily identifiable type of maintenance, often referred to as *repairing*, the correction of faults found after the product is deployed.

Adaptive Maintenance is a reactive activity, that envisages to make software usable in a changed environment [EFK+03]. Consists in adapting software interaction with new software, like operating systems, database management systems or hardware systems (input and output devices), machines... Software must also be adapted to new business rules and work patterns, governmental policies, etc.

Perfective Maintenance foresees the improvement of software, either in an increase in performance or improvement in requirements [EFK+03]. It ensures the evolution of software by improving attributes like usability, efficiency, dependability, reliability, and security. Also provides the improvement of maintainability affecting factors like testability and understandability, reducing the effort in all the maintenance efforts.

Preventive Maintenance is performed in order to increase its Maintainability [EFK+03]. It is a response to the negative effect of the other three activities, the increase of system complexity that reduces the maintainability. Preventive Maintenance concerns activities such as updating documentation and increasing the software's modular structure and code optimization. It is done in order to reduce the useless work by making the programs better structured and easier to understand.

The effect of preventive maintenance should be (as seen in Figure 3.1, in the moment signaled with P') to increasing the software maintainability to a level close to the initial maintainability of a product, and if possible even better, and keeping the maintainability at a higher level as time passes (compare Figure 3.1 with Figure 2.4 in Page 11)

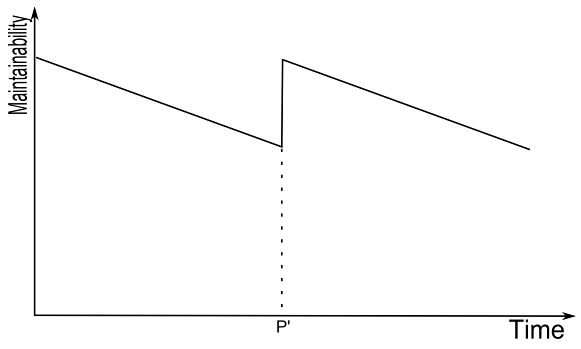


Figure 3.1: Maintainability over Time, with the effect of preventive maintenance

It is important to note that some authors [oEE98] consider preventive maintenance to be solely oriented at preventing problems in the foreseeable future and not as a maintainability increase effort.

We believe that an increase in maintainability is a step taken forward in the prevention of problems and therefore include this predictive activity as a sub-activity of this type of maintenance.

3.1.2 Maintenance Processes

To put in practice any of these types of maintenance some processes must be highlighted as essential. These processes are fundamental in the pursuit of the understanding of software and in increasing its maintainability.

Reverse Engineering is a process through which representations of a system are created through the analysis of the existing components code and their relations with other systems [EFK+03] when the only reliable representation of the software is its code [oEE98]. The representation resulting of this process is normally high-level documentation, but can also be lower-level documentation and even complete recodings. It is used as a mean to build or increase system understanding, mainly in preventive maintenance when documentation is scarce or outdated and also in adaptive maintenance, normally for migration between platforms.

Reuse and reusability goals during maintenance are “to increase productivity, increase quality, facilitate code transportation, reduce maintenance time and effort, and improve maintainability” [EFK$^+$03]. It applies to both processes, that can be reused throughout all the maintenance project, to personnel, whose experience could be reused, and also to product, whose data formats design and implementations can all be of use in new or re-engineered modules, packages or simple methods.

Re-engineering is the process by which a system is replicated. It consists in a reverse engineering followed by a forward engineering process [oEE98] normally in a selective way that promotes the reuse of code. It is also the process through which an old system is brought up to current standards and to newer technologies support. Although this process normally occurs in perfective maintenance, it is also possible to apply some corrective maintenance within re-engineering as the recoding is done.

3.2 Software Metrics

Accurate measurement is a prerequisite for all engineering disciplines, and software is not an exception [RLL08].

Software Management decision making process, like decision making in any kind of management, requires measures for a well-founded comparison of different plans or statuses.

A Software Metric must have a viewpoint that allows the code quantification to be interpreted for better decision making, and by summarizing software status in meaningful measures, useful to quantify possible improvements or occurred deterioration to the code.

These quantifications can be helpful for project’s goals definition, cost analysis and return evaluation and also for staff productivity and code quality analysis (for both evaluation and improvement needs).

Amongst the most common software metrics (for the OOP) are the ones how measure simple facts about software like:

- LOC (Lines Of Code) counts the lines of code of a class.
- NOM (Number Of Methods) is the methods in a class.

Other, more complex metrics, that require analysis of the codes meaning, can give better insight on the codes complexity and on bad design, like:

- CBO (Coupling Between Object classes) is the number of classes to which a class is coupled [Chi94].
- DIT (Depth of Inheritance Tree) is the maximum inheritance path from the class to the root class [Chi94].
- NOC (Number Of Children) is the number of immediate subclasses subordinated to a class in the class hierarchy [Chi94].
- RFC (Response For a Class) is the set of methods that can potentially be executed in response to a message received by an object of the class [Chi94].
- WMC (Weighted Methods per Class) is the sum of weights for the methods of a class [Chi94] (also know has Cyclomatic Complexity).
- Halstead measures, a series of metrics that defined problem lenght, vocabulary, volume, difficulty and effort [Mac09].

Another metric, very useful in this context, is the Maintainability Index, available in Visual Studio 2008 Team Suite. It is a relative (percentage) measure that uses three other metrics (previously referred Halstead Volume, Cyclomatic Complexity, and the number of lines of code) [Mor09]. Results under 20% indicate that the code in question has a really low maintainability.

3.3 Code Cloning

Code cloning makes software maintenance harder. A code clone is a portion of code that is similar or identical do another portion(s) of the source code. Amongst the reasons for the introduction of code clones are reusing by ‘copy-and-paste’, intentionally repeating code for (often empty) customization to new contexts or for performance reasons, failure in identifying/using abstract data types, or simply by accident [Bie98]. All of these practices should be avoided, in favor of a reusing library functions rather than cloning code.

Code cloning detection methods vary from techniques like simple String Matching, that address code syntax, to more advanced ones using abstract syntax trees or graph representations of code, that are semantic oriented [Eil05].

3.4 Design Patterns

Design Patterns are a collection of solutions for recurrent design problems in particular contexts [EGV98] in order to ease code re-use. Patterns are cataloged as a reference to software engineers to solve their design problems. Making use of a widely recognized catalog of design solutions enables better future understanding of the design and consequently of the code produced.

Design Patterns solve many of the OOP problems in different ways [EGV98]. Design Patterns help finding appropriate objects to decompose the system in, as well as helping to choose what is encapsulated inside each object. Patterns also help to specify object interfaces by help to defining the way objects interact with each other. They also help specifying object implementations and use relationships like inheritance and delegation. By helping to achieve patterned designs in the previous poins design patterns help putting reuse mechanisms to work and ultimately designing with anticipation to future requirements [EGV98].

The “Gang of Four” separated patterns in the following groups [EGV98]:

- Creational Patterns - for object creation
- Structural Patterns - for object organization
- Behavioral Patterns - for algoritms and responsability assigning between objects

3.5 Refactoring

Refactoring has two definitions, as a noun (*refactoring*) and as a verb (*to refactor*).
As a verb, to *Refactor* is an activity that aims to “change the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior” [Fow99].

In order to *Refactor* one applies *Refactorings*.

Refactoring, as a noun, is “the removal of duplication, the simplification of complex logic, and the clarification of unclear code” [Ker04]. A refactoring is a single action that makes one modification to the internal structure of the code.

To *refactor* implies the usage of a collection of mechanic instructions (*refactorings*) that are usually applied in small steps towards the goal of displaying the same behaviors but with clearer code or with better performance, and meanwhile testing every step taken, or else one can drive himself towards different outcome than the original software.

3.5.1 Motivations to Refactor

Amongst various reasons for recoding the most common are [Ker04]:

- Ease the addition of new features (and code)
- Improve the design of existing code
- Increase understanding of the code

- Make coding less annoying

3.5.2 Time to Refactor

Because refactoring improves the design of software and the overall quality of code, one could be drawn to think that the first thing to be done to any software would be to refactor. Obviously, in a schedule-driven industry to *waste* time coding for no functional change is not an option.

Three moments are good to apply the refactoring techniques to some specific code, when the following situations apply [Fow99]:

- While adding a new feature;
- When a bug fix is needed;
- While a code review is done.

While adding a new feature "is the most common time to refactor" [Fow99]. It helps understanding the code, specially when there is a bad design, and therefore helps to write new code or reuse existing one to add the desired features.

When a bug fix is needed is good time to refactor because the application of the refactoring techniques in themselves helps to better understand the code surrounding the defect and identifying it.

While a code review is done refactoring is also a good help to understand the code at hand and it also serves the main purpose of a code review as many suggestions for better coding will arise.

3.5.3 Anti-Patterns: the Reason to Refactor

There is a long list of anti-patterns that call for refactoring. The solutions presented for each of them are better explained in subsection 3.5.4 Common Refactorings. Some of the most common anti-patterns are the following [Fow99]:

Duplicated Code is easily identifiable when the same code structure is identified in different parts of the code. It creates a problem when there is need of a change in a feature that is used in different parts of the code but should always be done the same way. It is found either inside one class, in sibling subclasses, or even in apparently independent classes. It is normally resolved with the *Extract Class* refactoring.

Long Methods have been recognized has hard to understand since the early days of programming. The flow of a long method should be split to small sub-methods that are accurately named to make the original method's flow easy to understand without the need to constantly check what a sub-method does. It is resolved with *Extract Method*, sometimes associated with *Replace Temp with Query*.

Large Classes (also known as "Blobs") also tend to get over-complicated and hard to understand. A class should represent system entities, and for the sake of understandability they should be small. If a class is to large it is probably withholding more entities than it should. They're also a starting point for other anti-patterns like Duplicated Code and Divergent Change. The most common solution for Large Classes is to use *Extract Class*, *Extract Subclass*, or even both, whichever adapts best to the relations between entities.

Long Parameter Lists tend to get very hard to read and understand. In the OOP there is no need for all the information to be passed to methods as arguments. Information can be requested to objects, either created in the scope of methods or passed arguments. One of the existing solution for this anti-pattern is to use *Introduce Parameter Object*.

Feature Envy occurs when a method makes more use of information in another class than from the class the method itself is in. It is normally solved using *Move Method*.

3.5.4 Common Refactorings

The usage of refactorings is a solution to anti-patterns (see subsection 3.5.3) regularly found in code that was done in tight schedules and without a proper design.

It is important to understand that there is not a direct relation between a Refactoring and an anti-pattern. The proposed solution to resolve anti-patterns are a sequence of refactorings, that are gathered and sequenced in a certain way to remove specific problems from code in order to respect the desired design and keep the system behavior.

Refactorings can be separated in types, depending on what they affect (this separation was used by Martin Fowler in [Fow99]),

- Composing Methods - works within an entity
- Moving Features Between Objects - works between entities
- Organizing Data - works with variables and subclasses
- Simplifying Conditional Expressions - works with conditions
- Making Method Calls Simpler - works with method calling

- Dealing with Generalization - works with inheritance and delegation

Refactorings themselves are not singular actions, but also a sequence of instructions to flow, and although they are as atomic as possible, at times, some refactorings are done using other simpler refactorings. Some of the most common refactorings are [Fow99]:

Extract Method consists in removing a code fragment from one method, grouping it in a new method. The new method should be carefully named so that the code fragment functionality is easily identified.

Replace Temp with Query consists in replacing a temporary variable, that is immutable and merely holds the result of a simple call or expression, with a method that returns the same result. It is often used to ease other refactorings, like Extract Method.

Introduce Parameter Object is used to replace groups of parameters that are normally passed together and that might be logically related with objects that contain the same information and methods to access it.

Extract Subclass is used to create a subclass to a class when this is a better design solution, such as when a class is using only part of its resources in some specific and clearly identifiable situations.

Move Method is used to move a method from one class to another when it is desired for the class to be in the second class.

3.5.5 Refactoring to Patterns

In order to achieve a clear design, with usage of patterns, that make the code understandable and more pleasant to work with, refactoring must be taken one step beyond.

As noted by Kerievsky, reference literature on both Design Patterns and Refactoring recognizes the former as a natural mean to reach the first [Ker04]. Having a “Refactoring to Patterns” approach is using the refactorings previously described, in the scope of higher level refactorings. These high level refactorings are thoroughly oriented to achieving a design that respects patterns as much as possible [Ker04].

It is important to understand that a refactoring can take different directions: *to*, *towards* or *away* from pattern [Ker04]. The difference between *to* and *towards* is that the first is pattern-directed and the second is pattern-directed and only stops when the pattern is reached. Not refactoring towards a pattern is not wrong: sometimes reaching a pattern simply does not payoff and can be even worse than moving away. The most important is to walk into understandability.

3.6 Summary

Maintenance of a software application is a complex process that normally has big costs associated to it. This condition is usually connected to defective software development in the previous phases of the software life cycle that result in scarce design definition, poor documentation, and unintelligible code.

Ideally, software should be designed (and coded) according to patterns that represent solutions to common design needs. This approach promotes the understandability and ease of feature changing and addition required by a maintenance process.

The usage of an organized approach like refactoring is the ideal method to evolve from a messy application with lots of anti-patterns to a well designed application that applies design patterns, is easily understandable and create better documentation in the process.

The goal is simple: increase maintainability.

Chapter 4

Solution Specification

iFAct was a project in clear need for an increase in maintainability. The solution proposed in this chapter is a design that is intended to be used in the recoding of iFAct (both Lab and Web applications) and was developed taking in account that the project will be recoded in parts, as the need or possibility to do it arises. It was also considered that the goal was to create a design that applies patterns and that can be achieved through the refactoring of iFAct's existing code.

Taking in account the need to have a very understandable and easy to apply design, a number of patterns were studied in order to choose an appropriate solution to the new iFAct design.

4.1 Presentation Patterns

The patterns studied as options for the new architecture were:

- Model-View-Controller (MVC)
- Supervising Controller (SC)
- Passive View (PV)

4.1.1 Model-View-Controller

The MODEL-VIEW-CONTROLLER is the most *en-vogue* design pattern, specially for web applications. It has achieved a huge popularity with the advent of fast-development frameworks like Rails for Ruby, a nearly uncountable number for PHP: Hypertext Processor (PHP), and a lot for a number of other languages [con09], to the point that Microsoft® itself recently launched ASP.NET MVC Framework for .Net Framework 3.5. It is a solid solution for isolation of Business Logic from User Interface. Despite its success in web-apps MVC is not used exclusively for this purpose, and it was, in fact, firstly used with Smalltalk [Ree03].

MVC is normally depicted as seen in Figure 4.1, the control flow normally goes as follows [con09]:

1. The user interacts with the user interface in some way.

2. The controller handles the input event from the user interface.

3. The controller notifies the model of the user action for the necessary change in the model's state to be taken.

4. A view uses the model indirectly to generate an appropriate user interface. The view gets its own data from the model.

5. The user interface waits for further user interactions, which restarts the cycle.

The "indirect usage" of the Model by the View referred in the pattern description (represented in Figure 4.1 with the dashed line) is the Observer pattern.

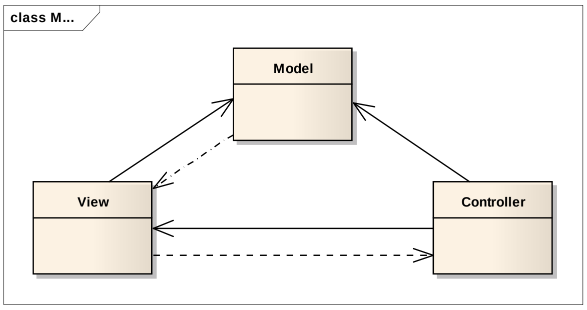


Figure 4.1: Model-View-Controller

4.1.2 Supervising Controller

The SUPERVISING CONTROLLER is a variation of Model-View-Presenter. It is a presentation pattern that behaves in a way that is similar to the MVC. There are many relations between all the intervenient classes. It is characterized by the fact that the Controller (also named Presenter) is aware of the changes in the Model, making use of the Observer pattern, like the View does in both MVC and also here in SC.

In Figure 4.2 the dashed-lines once again represents the Observer pattern.

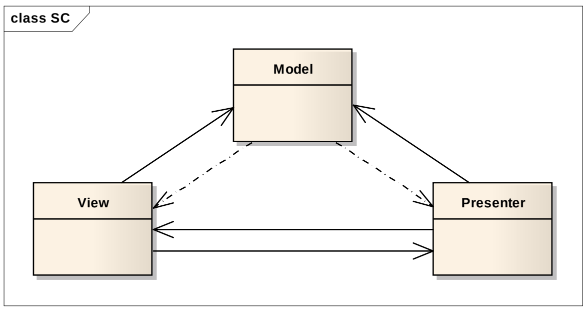


Figure 4.2: Supervising Controller

4.1.3 Passive View

PASSIVE VIEW is another variation of Model-View-Presenter. It is characterized by the simplistic approach to separation between the layers, yet it is the one that isolates the layers the most. The view is absolutely passive, limiting itself to presenting the user interface and diverting events to the controller.

The whole work is done by the Presenter. It receives the input from the user interface, retrieves the required information from the Model, and then updates the View accordingly.

The high level of isolation between the layers, depicted in Figure 4.3 increases the testability of the classes because the interfaces are much simpler and the triggering of events is also more understandable.

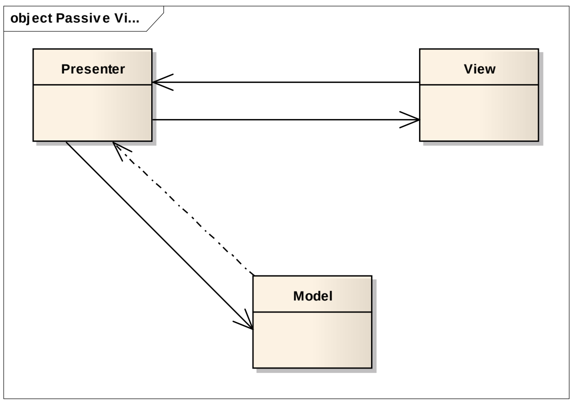


Figure 4.3: Passive View

It is also more practical to make a controller work for both Web and Lab application with the simpler interfaces implemented with the Passive View pattern.

The chosen Presentation pattern for the refactoring was the Passive View. The main advantages over the other possibilities are the better testability and the fact that the simpler layering allows to use less effort demanding refactorings.

These two criteria weighted very much because testability and development speed are important for a project with several stability problems that is in active corrective maintenance.

The evolution to a more developed presentation pattern can be done in a latter stage, when the maintenance is more oriented to perfective or preventive maintenance than to corrective maintenance, which is not the case with iFAct.

4.2 Delegation

Another goal of this project was to remove code duplications. Besides the obvious duplication of parts of code amongst methods, there is also the need of sharing methods between controllers.

Between the choices of Delegation and Inheritance it was chosen that the Delegation pattern would be the best option.

Scalability and the freedom level permitted by Delegation were the criteria taken in account for this choice. Delegation allows the usage of methods from an infinite number of classes, and inheritance would force some unnecessary information to be place in a controller.

In Figure 4.4 it is introduced the concept of Global Controllers (represented by *GController_X*). These objects are the common method holders and the main mean to reduced code repetition.

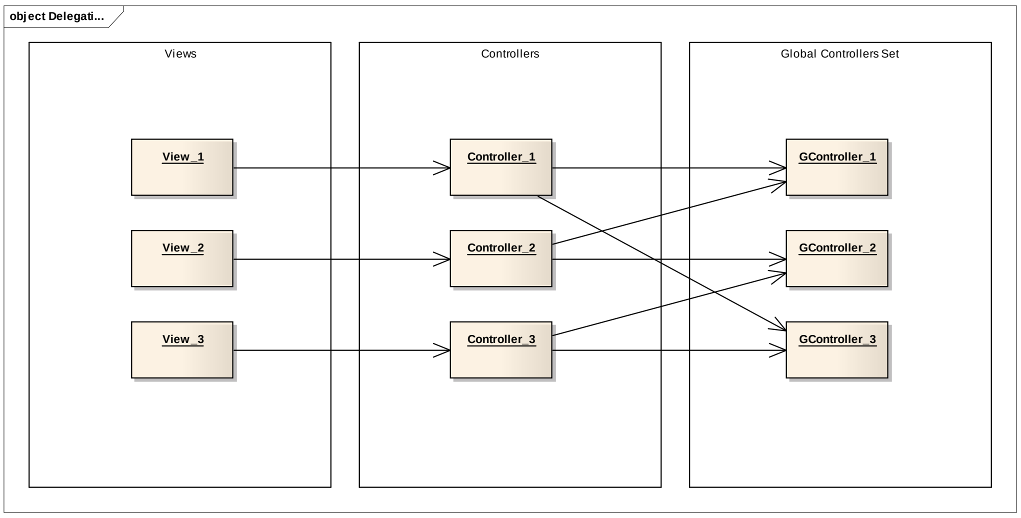


Figure 4.4: Delegation

Notes

In addition to the choice of a presentation pattern it is important to note some details.

The mode of connection to the database will be kept, as most of the connections are made in a way that can be considered correct: a call to a stored procedure. The connections will be made by the model and the database will be treated as a black box.

The relations between interfaces (like the creation of interfaces of another type, passing of data) are made by the controllers.

4.3 Roadmap to Design Application

The first approach taken into iFAct's recoding was encouraged by the belief that the code was more stable, better structured and easier to understand than it was in fact. The first attempt to refactor some of iFAct's classes were made only to find out that there were very bad practices like windows forms keeping the data of the model entities. meaning the same classes were responsible for the interface and the connection with the database and that they were not holding any kind of other objects to represent the business model.

This unexpected difficulty led early development to a dead-end in the first approach to the code, but the experience gained from it had its benefits: the bottom-up approach created knowledge and understanding on the applications business model that were fundamental to the project's continuation.

iFAct's recoding process should undergo the following stages:

- Business Model Analysis;
- Design Orientation Definition, according to Business Model, proposed Presentation Patterns and existing features;
- Create *dummy classes* for the Passive View implementation;
- Fill the *dummy classes* by refactoring the attributes and behaviors from the original "View" to the correct class in the Presentation Pattern, and if applicable refactor the behaviors and attributes.

4.4 Summary

After a study of some Presentation Pattern, Passive View was chosen as the new pattern to address iFAct's needs. It will allow to have a clearly separated and easy to understand three-layered application with a high testability level and seamless implementation of controllers for different views. To avoid code repetitions of common usage features in the controllers it should be used a Delegation scheme, where global controllers are given the tasks by the controllers more directly associated with the views. In order to make the refactoring, one should first analyze the business model of the functionalities it is working with and then, with a loosely defined design based on the business model, start creating the classes necessary to implement Passive View and later refactor the methods to the correct classes and in the correct form.

Chapter 5

Proof of Concept

The solution proposed for iFAct Recoding in the previous chapter was tested in a proof of concept that involved the recoding of iFAct's Product Management features.

Some concepts should be understood in order to understand the business logic of the proof of concept. A *products* is, a final article. Every product is constituted by it's *components*, e.g., a computer (product) that has a collection of components (a motherboard, a keyboard, a network driver, etc.). Every *component* can also have their one child *components*, and this can extend infinitely. A *product* can never be the child of another *product* or *component*.

In iFAct a *product* has so many common data with a *component* that both should be considered as specializations of *items* (later it can be understood that this is actually useful in defining the business model). Every time we refer to *Item* it should be understood that it is either a *Product* or a *Component*.

5.1 Original Status

iFAct's Product Management features have no interface with other software outside of the iFAct's domain. The uses cases involved are accessed in iFAct's applications and all data involved is retrieved and updated to iFAct's Database.

Much like everything else in iFAct, the Product Management is implemented in a sole class, a Windows Forms that is responsible for creating the interface, all the business logic and also for most of the data access layer.

The use cases for iFAct Product Management are depicted in Figure 5.1.

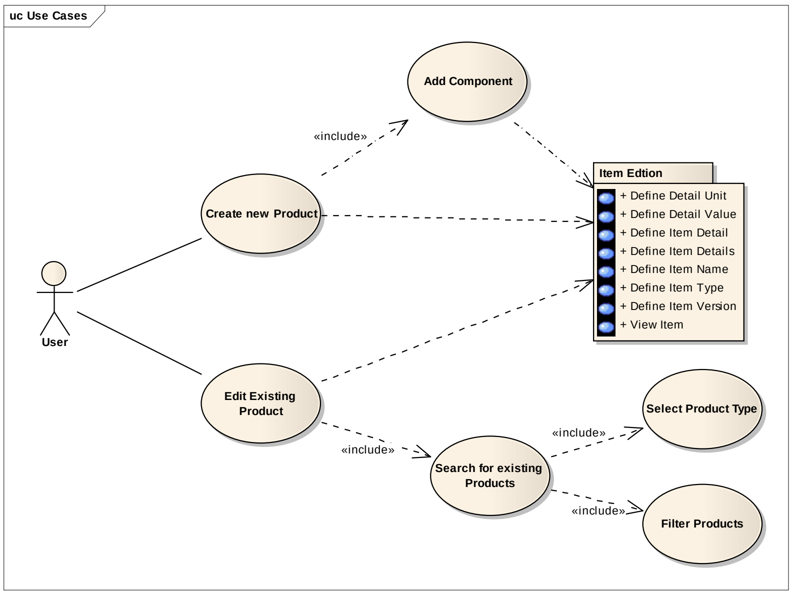


Figure 5.1: Product Management Use Cases

In Figure 5.2 it is possible to see the use cases required for the item edition.

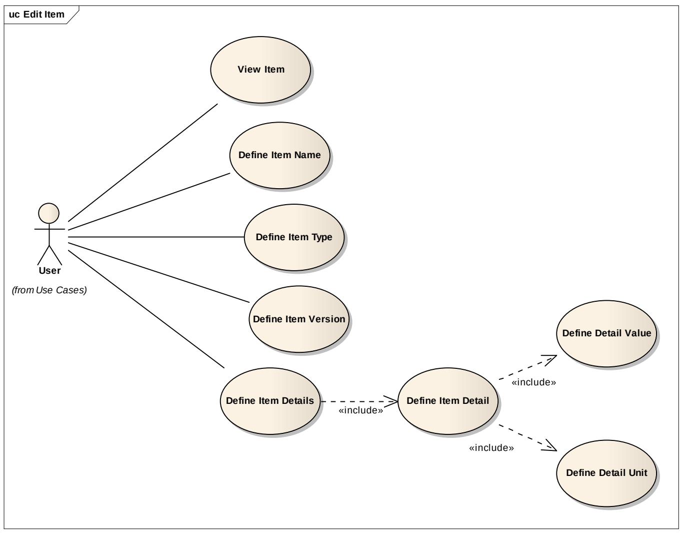
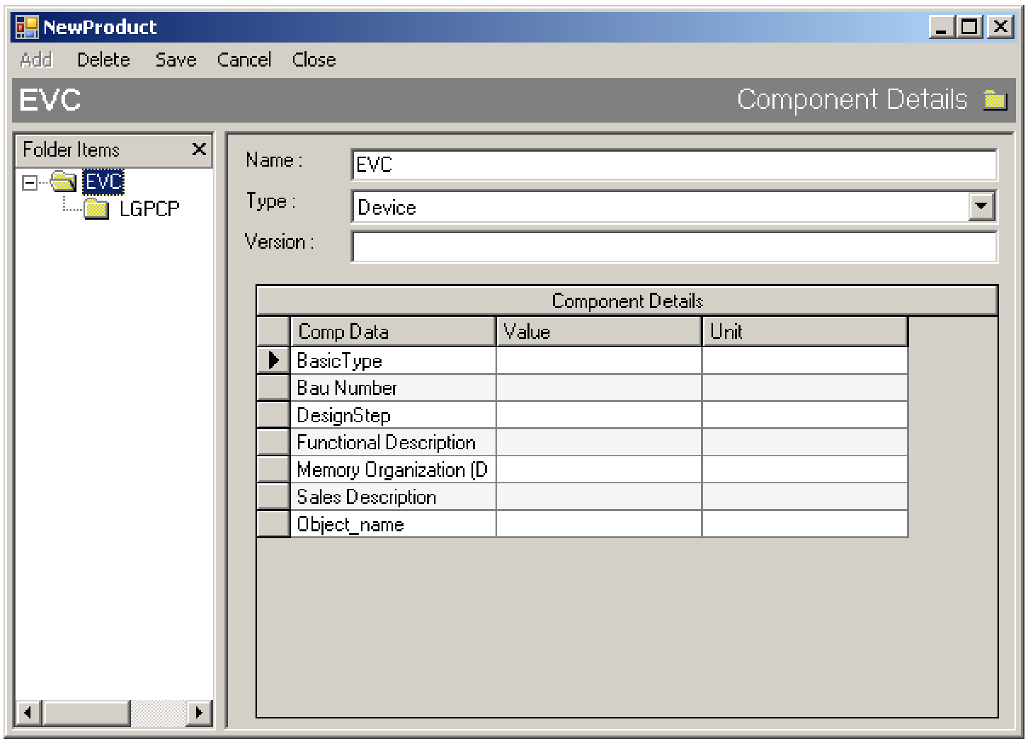


Figure 5.2: Item Edition Use Cases

iFAct had the following forms for the given use cases:

- New Product - to create a new *product* and edit its items
- Search Product - to search for *components* to add to new products
- Product Explorer - to see, edit and delete existing *items*

New Product is the form (seen in Figure 5.3) to create new *products* and to create or add its *components* to it.



Name : EVC

Type : Device

Version :

Component Details

	Comp Data	Value	Unit
▶	BasicType		
	Bau Number		
	DesignStep		
	Functional Description		
	Memory Organization (D		
	Sales Description		
	Object_name		

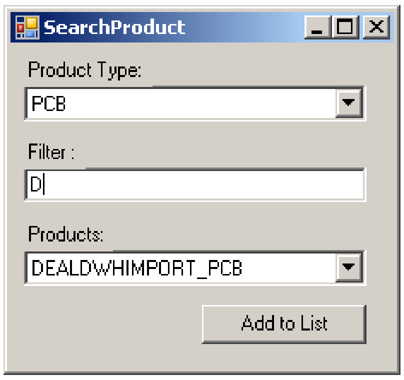
Figure 5.3: New Product Form

The New Product form includes the following actions

- Add a new *item* (*BUG!* should be a component) to the *product*;
- Add an existing *item* (*BUG!* should be a component) to the *product* (through the Search Product Form);
- Remove *component* from a *product*;
- Select which item to edit (from the TreeView on the left)
- Edit the name of *product* and of its *components* (as a string);
- Edit the type of *product* and of its *components* (a selection from a DropDownList);

- Edit the version of *product* and of its *components* (as a string);
- Edit the value and of *item* details (through a DataGrid)
- Save the *product* and respective *components*;
- Cancel *product* creation;

Search Product is the form (seen in Figure 5.4) called from the *New Product* form to search for new a *item* (*BUG!* should be a component) and to add them to a new *product*.


SearchProduct

Product Type: PCB

Filter : D

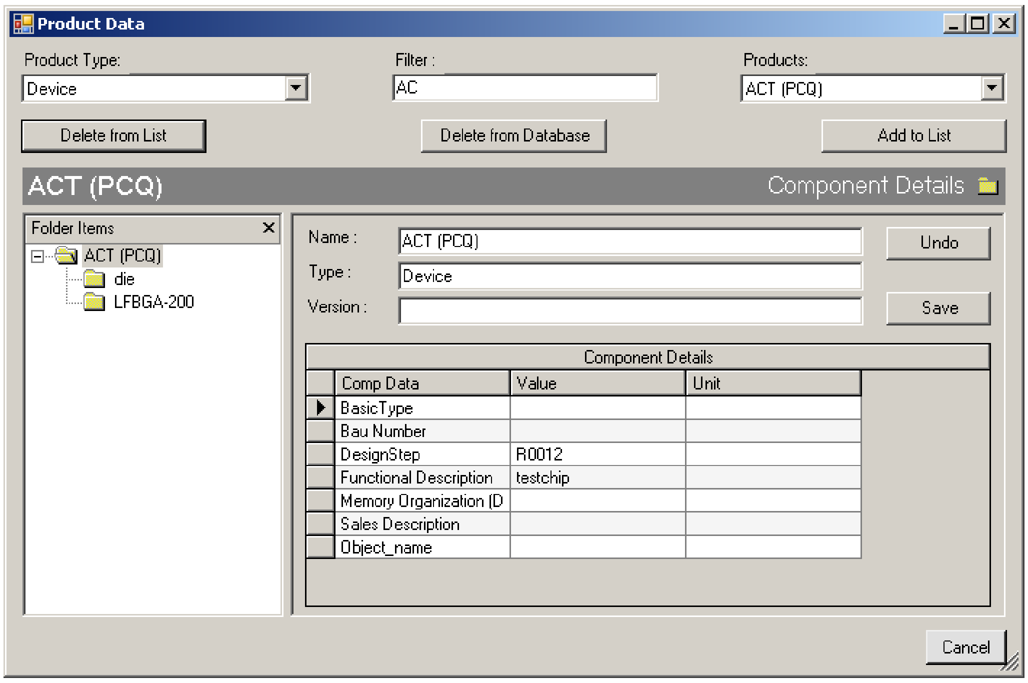
Products: DEALDWHIMPORT_PCB

Figure 5.4: Search Product Form

The Search Product form includes the following actions

- Select the *item* (*BUG!* should be a component) type to search for (a selection from a DropDownList);
- Write a filter string for the name to trim down the list of *items* (*BUG!* should be components)
- Select the *item* (*BUG!* should be a component) (from a DropDownList) to pass on to the New Product form;
- Cancel action.

Product Explorer is the form (seen in Figure 5.5) that allows to search for *items* (*BUG!* should be products) to edit its details, including existing *components*.



Product Type: Device

Filter : AC

Products: ACT (PCQ)

Folder Items:
- ACT (PCQ)
  - die
  - LFBGA-200

Name : ACT (PCQ)

Type : Device

Version :

Component Details

Comp Data	Value	Unit
BasicType		
Bau Number		
DesignStep	R0012	
Functional Description	testchip	
Memory Organization (D		
Sales Description		
Object_name		

Figure 5.5: Product Explorer Form

The Product Explorer form includes the following actions

- Select the type of *item* (*BUG!* should be a product) as a filter to trim down the list of *items* (from a DropDownList);
- Write a filter string for the name of the *products* to trim done the list of products (as a string);
- Select the *item* (*BUG!* should be a product) to edit (from a DropDownList);
- Add the *item* (*BUG!* should be a product) to the edition list;
- Delete the product from the edition list;
- Delete the product from the Database;
- Select which *item* to edit (from a TreeView on the left);
- Edit the name of *product* and of its *components* (as a string);
- Edit the type of *product* and of its *components* (as a string [*BUG!* should be a selection from a DropDownList]);
- Edit the version of *product* and of its *components* (as a string);

- Edit the value and of *item* details (through a DataGrid)
- Save the edited *product* and respective *components*;
- Undo the changes made to the *product* and respective *components*;

It is interesting to note the several defects (signaled in the list of use cases) that are related to a business logic detail that is not explicitly documented and therefore as been programmed “carelessly”.

5.2 Business Model

The concept of a *product* in iFAct is intimately related to the former’s business model.

It is important at this point to recall that a *product* has so many common data with a *component* that both should be considered specializations of *items*.

5.2.1 Attributes

A *product* is composed by *components* which themselves can also be composed by other *components*. Considering this, a *product* is a tree. A *product tree* is unlimited in its depth.

All *items* have:

- a *name* (required);
- a *type* (required);
- a *version*;
- a parent (only for the components);
- a list of details;
- a list of children;

The types are different for *products* and for *components*.

The list of details of an *item* is a simple list in which each entry is a triplet. This list name is either *Product* Details or *Component* Details, accordingly to the kind of *item*. The triplet is composed of the fields:

- Data;
- Value;
- Unit;

The list of children is simply a list of the *components* that constitute this *item* (*products* can not be children).

Because of all these common elements the *item* abstract class will implement almost all the features, leaving the few differences of behavior to be done by the extending sub-classes *products* and *components* as shown in Figure 5.6.

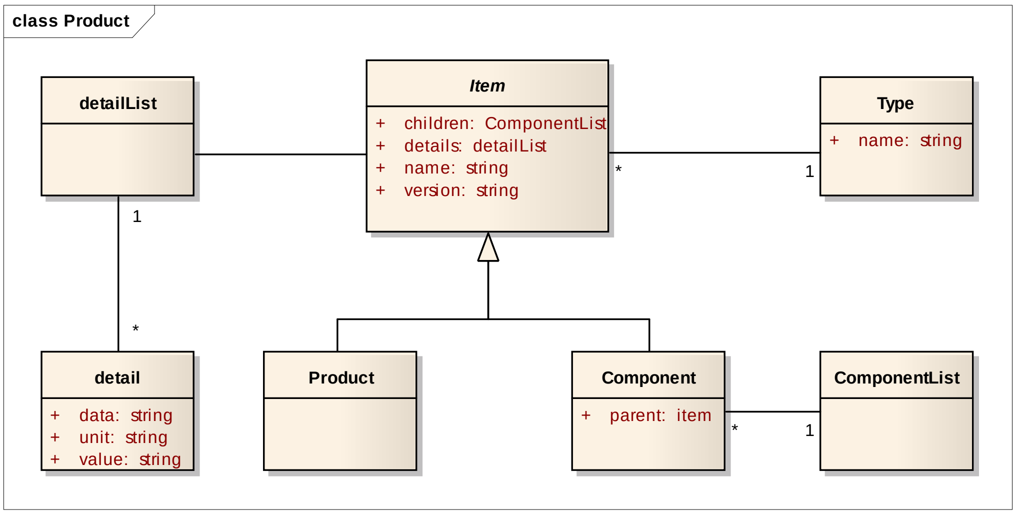


Figure 5.6: Product Business Model

5.2.2 Behaviors

Product and Component behaviors should be split into two types: those responsible for transactions with the database and others responsible for keeping the model update with the changes made in the presentation before any database operations.

Database transactions are traditional Create Retrieve Update and Delete (CRUD) operations triggered by the events with typical names like “get”, “save”, “update” or “delete”. Some of these methods need to be recursive as the database stored procedures are not ready for receiving a tree of components.

The model updating methods have similar goals, but instead of getting data from the view, the model updates with the data received from the controllers’ calls.

5.3 Design

In Figure 5.7 it can be seen the high level design of the solution for the product management. The three forms and the web view are shown in the View area. The views instantiate the corresponding Presenters, which in it's turn instantes the Model class for database access and runtime data management.

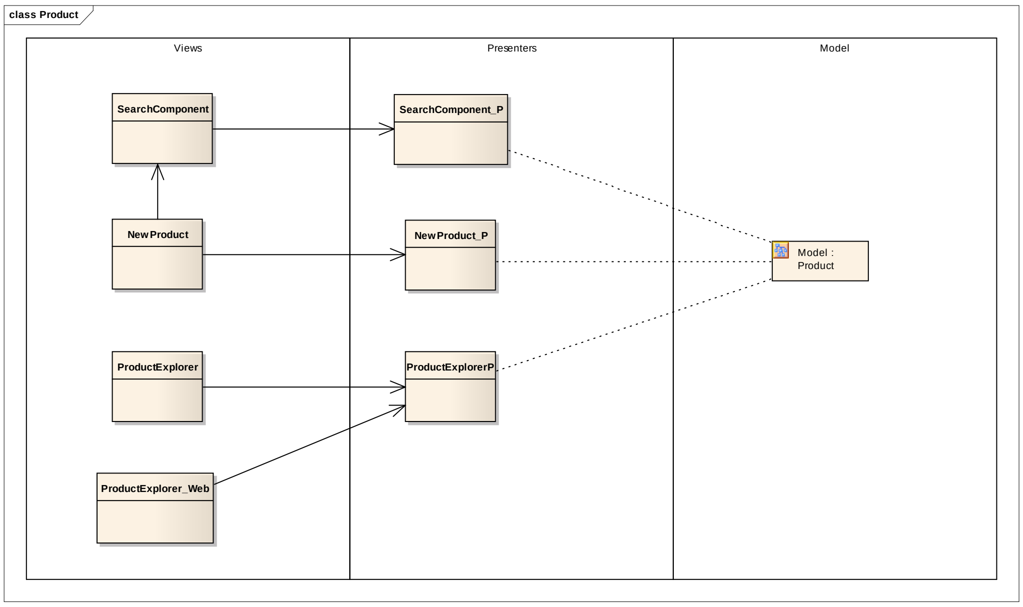


Figure 5.7: Product Management High Level Design

In Figure 5.8 the design in depicted in more detail. Both the View and the corresponding Presenter have well defined interfaces, through which interaction is made. Both objects only have knowledge of the interface of the other. Adequate interface definition allows the presenter to control the view without any knowledge of how the user interface is implemented.

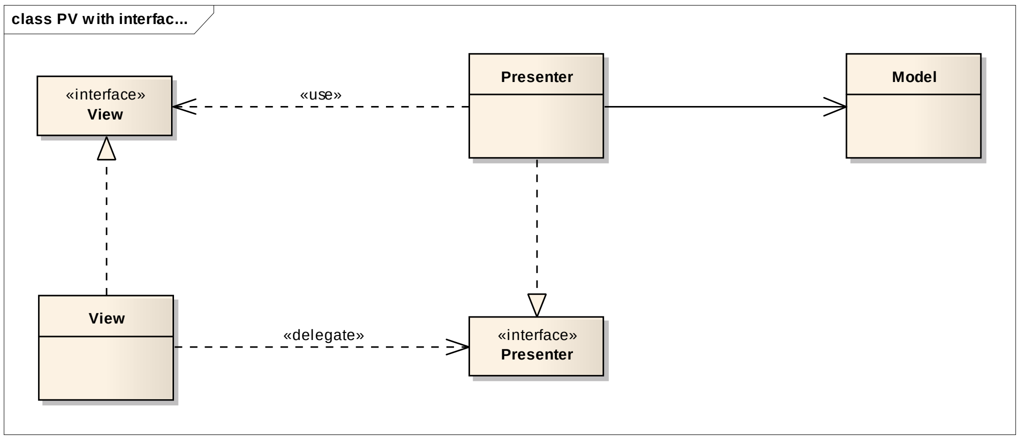


Figure 5.8: Presenter-View Interfaces

The Views' interfaces consist in simple orders (e.g. *UpdateName* or *SignalNameError*), and a event registering access. These can be implemented in different ways for any kind of view that might be desired (e.g. Windows Form, Web App, Console App, Mobile App).

The Presenter' interfaces require the definition of a delegation method that is registered to handle the events in the necessary Views.

It should be noted that ten bugs identified in iFAct's Product Management behavior, in section 5.1, were fixed during the recoding .

5.4 Experimental Results

The first attempt to refactor was done without the Business Model definition which resulted in total failure. The final, previously described approach was defined based on the experience acquired from this underachievement. Table 5.1 shows a comparison between the original and the recoded Product Management (with the same metrics as the ones used in Table 2.1)

Metric	Product Management	Product Management Recoded
Number of Classes	4	16
Methods	35	41
Lines of Code	2327	1900
Max Complexity	81	29
Average Complexity	4,5	2,9
Maitainability Index	13	33

Table 5.1: Code Metrics for Product Management Recoding

The new implementation has a much bigger number of classes, increasing from 4 to 16 as a consequence of the separation of the Presenter and the Model, which in itself has 8 classes, from the View. This is a natural consequence of what is believe to be a better and much more intelligible design There was also an increase of 15% in the number of methods, resulting from some delegations and also some long method spliting. The code duplication reduce the size of code in about 20%. The increase in the Maintainability Index indicates that the work done was successful, raising the index from a warning level (between 10% and 19%) to a acceptable level (above 20%) [Mor09].

5.5 Summary

The most important part of the refactoring in iFAct, and the base for a sucessful recoding is the concrete knowledge of the Business Model to be worked with. From the Business

Model one can apply the proposed design, Passive View, and start refactoring the existing views to fill the design keeping the data flow as most as possible.

Although the increase in classes and that it was impossible to test the code in maintenance environment, it is believed that the refactored version of the Product Management is more maintainable and that the new design can be successfully implemented.

Chapter 6

Conclusions

iFAct Recoding was a preventive maintenance effort taken through the course of sixteen weeks with the goal to increase maintainability of iFAct by studying a way to avoid code duplication, structure the code better, and create an adequate level of modularity.

Software maintainability is not easily risen, specially without proper documentation and after several programmers have worked on the code making it untidy and very unintelligible like it was the case with iFAct.

The preventive maintenance work done during this Master's Project had a lot of unnecessary efforts that could have been avoided if the project had had more careful treatment of the software documentation and code understandability through the products life cycle.

Refactoring techniques were very important during the recoding process as they were the cornerstone of the understanding process of the code, albeit only on a second approach to the project it was possible to achieve a successful outcome. This understanding could have never been achieved without the *bottom-up* approach used in the beginning of this master's thesis project.

6.1 Proposed Solution Evaluation

It was only possible to test the proposed solution through the development of a Proof of Concept. Because of the project's nature, it has been impossible to evaluate the solution in practice by analyzing maintenance statistics of refactored code respecting the new design. In spite of that difficulty, in our opinion the solution achieved respects thouroughly the principles present in literature for the defined goals an it is believed the implementation of the new design to the whole project will increase its maintainability. Therefore the result achieved from the work developed is considered satisfatory.

6.2 Future Work

The implementation of the proposed solution to the whole project is of interest because of the prospective increase in maintainability to a project that has such a big effort in corrective and preventive maintenance. There is also the perspective to see if the recoding process evolve and adapt even better to the project with the lessons learned through the real world application.

References

[AG09a] Infineon Technologies AG. Infineon technologies, 2009. http://www.infineon.com/ consulted 2009 April 3.

[AG09b] Infineon Technologies AG. Product information, 2009. http://www.infineon.com/cms/en/product/index.html consulted 2009 June 2.

[Ass09a] Semiconductor Industry Association. Frequently asked questions, 2009. http://www.sia-online.org/cs/industry_resources/individual_faq?siafaq.id=1 consulted 2009 June 3.

[Ass09b] Semiconductor Industry Association. Frequently asked questions, 2009. http://www.sia-online.org/cs/industry_resources/individual_faq?siafaq.id=3 consulted 2009 June 3.

[Ass09c] Semiconductor Industry Association. Industry fact sheet, 2009. http://www.sia-online.org/cs/industry_resources/industry_fact_sheet consulted 2009 June 3.

[Bie98] Ira D. Baxter; Andrew Yahin; Leonardo Moura; Marcelo Sant'Anna; Lorraine Bier. Clone detection using abstract syntax trees. In IEEE, editor, *Proceeding of the ICSM'98*, November 1998.

[BSIG09] Inc. Bluetooth Special Interest Group. Basics, 2009. http://www.bluetooth.com/Bluetooth/Technology/Basics.htm consulted 2009 June 3.

[Chi94] C.F.; Chidamber, S.R.; Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476 – 493, June 1994.

[con09] Wikipedia contributors. Model-view-controller, 2009. http://en.wikipedia.org/w/index.php?title=Model%96view%96controller&oldid=298549125 consulted 2009 June 27.

[EFK+03] Kagan Erdil, Emily Finn, Kevin Keating, Jay Meattle, Sunyoung Park, and Deborah Yoon. Software maintenance as part of the software life cycle. Technical report, Department of Computer Science, Tufts University (TU), December 2003. http://hepguru.com/maintenance/Final_121603_v6.pdf consulted 2009 April 2.

[EGV98] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1998.

[Eil05] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, April 2005.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, sixteenth edition, 1999.

[Ker04] Joshua Kerievsky. *Refactoring To Patterns*. Addison Wesley, first edition, 2004.

[Leh80] Meir M. Lehman. Programs, life cycles and the laws of software evolution. In IEEE, editor, *Proceeding of the IEEE, Vol.68, No. 9*, page 1068, September 1980.

[Mac09] Virtual Machinery. The halstead metrics, 2009. http://www.virtualmachinery.com/sidebar2.htm consulted 2009 July 23.

[McC93] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 1993.

[Mor09] Conor Morrison. Maintainability index range and meaning, 2009. http://blogs.msdn.com/fxcop/archive/2007/11/20/maintainability-index-range-and-meaning.aspx consulted 2009 July 23.

[oEE98] The Institute of Electrical and Electronics Engineers. Ieee standard 1219-1998: Standard for software maintenance, 1998.

[Ree03] Trygve M. H. Reenskaug. Mvc xerox parc 1978-79, 2003. hhttp://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html consulted 2009 June 23.

[RLL08] Jonas Lundberg Rüdiger Lincke and Welf Löwe. Comparing software metrics tools. Technical report, Växjö University, Sweden, 2008.

[SA09] Critical Software SA. Critical software - company, 2009. http://www.criticalsoftware.com/ consulted 2009 April 2.

[SB07] Richard W. Selby and Barry W. Boehm. *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research*. Wiley-IEEE, reprint, illustrated edition, 2007.

[Sof09] Campwood Software. Sourcemonitor version 2.5, 2009. http://www.campwoodsw.com/sourcemonitor.html consulted 2009 June 27.

[Som07] Ian Sommerville. *Software Engineering*. Pearson Education, eighth, ilustrated edition, 2007.

Glossary

.Net Framework A sofware framework developed be Microsoft Corporation. 10, 24, 43

anti-pattern an easily identifiable (like a pattern) wrong design, normally originated from mis-oriented coding; also know as *bad-smell* in code. 7, 8, 19, 20, 22

ASP.Net Framework .Net Framework's web application framework. 10

Bluetooth A short-range wireless communications technology intended to replace the cables connecting portable and/or fixed devices while maintaining high levels of security. [BSIG09]. 2

C# (pronounced C Sharp) is a multi-paradigm programming language, part of the .Net Framework. 7, 10

DEAL One of the databases that withholds information about Infineon Products. 8

DWH One of the databases that withholds information about Infineon Products. 8

job A collection of tests to be performed as part of the failure or reliability analysis. 6, 9

maintainability A set of attributes that bear on the effort needed to make specified modifications

(ISO/IEC TR 9126-4:2004). 15

Observer pattern A design pattern where one or more objects (observer) register themselves in another (subject). Whenever a relevant change occurs in the subject, it warns its registered observers. The observers than take the actions defined in themselves. 24

Quasi7 A database that automatically submits jobs to the iFAct sytem. 8, 9

RealisRel Software application used at Infineon Technologies to assist reliability tests procedures. Automatically submits jobs to iFAct. 8, 9

Ruby an interpreted, Object-Oriented programming language. 23

SAP Quality Management An application from the German enterprise SAP. Automatically submits jobs to the iFAct system. 8, 9

silicon A chemical element which has the symbol Si and atomic number 14. It is the most common metalloid. It is a chemical element with atomic mass of 28.0855.. 2

Source Monitor Open Source Multi-Language Code Metrics Analyzer Application available at `http://www.campwoodsw.com/sourcemonitor.html`. 5