Mikołaj Olszewski

Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems

Turku Centre *for* Computer Science

TUCS Dissertations
No 161, August 2013

Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems

Mikołaj Olszewski

To be presented, with the permission of the Department of Information Technologies of Åbo Akademi University, for public criticism in Auditorium *Gamma* in the ICT Building in Turku, on the 22nd of August, 2013, at 12:00.

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5A, 20540 Turku, Finland

2013

Supervised by

Professor Ralph-Johan Back
Department of Information Technologies
Åbo Akademi University
Turku, Finland

Reviewed by

Professor Kai Koskimies
Department of Software Systems
Tampere University of Technology
Finland

Professor Tomi Männistö
Department of Computer Science and Engineering
School of Science, Aalto University
Helsinki, Finland

Opponent

Professor Kai Koskimies
Department of Software Systems
Tampere University of Technology
Finland

ISBN 978-952-12-2920-6

ISSN 1239-1883

A designer knows he has achieved perfection
not when there is nothing left to add,
but when there is nothing left to take away.

Antoine de Saint-Exupery

Abstract

Developing software is a difficult and error-prone activity. Furthermore, the complexity of modern computer applications is significant. Hence,an organised approach to software construction is crucial.

Stepwise Feature Introduction – created by R.-J. Back – is a development paradigm, in which software is constructed by adding functionality in small increments. The resulting code has an organised, layered structure and can be easily reused. Moreover, the interaction with the users of the software and the correctness concerns are essential elements of the development process, contributing to high quality and functionality of the final product.

The paradigm of Stepwise Feature Introduction has been successfully applied in an academic environment, to a number of small-scale developments. The thesis examines the paradigm and its suitability to construction of large and complex software systems by focusing on the development of two software systems of significant complexity.

Throughout the thesis we propose a number of improvements and modifications that should be applied to the paradigm when developing or reengineering large and complex software systems. The discussion in the thesis covers various aspects of software development that relate to Stepwise Feature Introduction. More specifically, we evaluate the paradigm based on the common practices of object-oriented programming and design and agile development methodologies. We also outline the strategy to testing systems built with the paradigm of Stepwise Feature Introduction.

Sammandrag

Utveckling av programvara är en besvärlig process som innebär betydande risk för produktfel. Därutöver är moderna datortillämpningar mycket komplexa. Av dessa orsaker är en välorganiserad process för programvaruutveckling av kritisk betydelse.

Inom Stepwise Feature Introduction - ett utvecklingsparadigm skapat av R.-J. Back - konstrueras programvara genom att funktionaliteten utökas i små successiva steg. Den slutliga koden får en organiserad lagerstruktur och kan enkelt återanvändas. Därutöver är interaktionen med programvarans användare och programvarans korrekthetsaspekter centrala element inom utvecklingsprocessen, vilket kontribuerar till hög kvalitet och funktionalitet hos den färdiga produkten.

Stepwise Feature Introduction-paradigmet har framgångsrikt tillämpats i universitetsmiljö i ett antal småskaliga projekt. Denna avhandling undersöker paradigmet och dess lämplighet i samband med konstruktion av stora och komplexa programvarusystem genom att fokusera på utvecklingen av två programvarusystem av betydande komplexitet.

Avhandlingen föreslår ett antal förbättringar och modifikationer som bör införas i paradigmet under utveckling och omstrukturering av stora och komplexa programvarusystem. Diskussionen i avhandlingen täcker olika aspekter av programvaruutveckling relaterade till Stepwise Feature Introduction. Mer specifikt evalueras paradigmet med avseende på praktisk objektorienterad design och programmering, samt på agile-utvecklingsmetodologi. I avhandlingen skisseras också en strategi för att för att testa system konstruerade med Stepwise Feature Introduction.

Acknowledgements

Even though a Ph.D. thesis has one author, it is a result of hard work of many people. I would like to take the opportunity to express my gratitude towards those who contributed to my work.

First and foremost, I am grateful to my supervisor, professor Ralph-Johan Back. I feel privileged to have been his Ph.D. student. I wish to thank him for his guidance and valuable feedback, as well as for trusting me to pursue my research independently. Moreover, as he is the author of the theory that I worked with during my research, my thesis would not at all be possible without him.

I would like to sincerely thank my opponent at the public defence, professor Kai Koiskimies, and professor Tomi Männistö, for taking the time and effort to review my thesis. Their valuable comments greatly contributed to the quality of the thesis and also allowed me to successfully overcome some of the problems by looking at them from a different perspective.

Professor Iván Porres was co-supervising my research at its early stage. He also coordinated the projects organised within Gaudí Software Factory, two of which I had a pleasure and an honour to supervise. I am thankful for your guidance and teaching me the importance of asking the right questions when starting any software development project.

I wish to express my gratitude and respect to professor Barbro Back for everything she did to help me start, advance and finish my research.

This thesis is, in large part, based on the results of a software development project that was a part of research project *BioTarget*. Participating in this cross-disciplinary collaboration was a remarkable experience, as it showed that software development techniques can help in solving problems that are very distant from computer science. Therefore, I would like to thank the leaders of research groups involved in the project, professors Jyrki Heino, Sirpa Jalkanen, and Mika Lindén.

My collaborator in the abovementioned project was Pasi Kankanpää, to whom I owe my most sincere gratitude. For the most part, the discussions we held concerned the project we both worked on. However, occasionally the topics were not that related, thus providing a much-needed refreshing break from everyday research.

Natalia Díaz Rodríguez, a talented programmer with an outstanding personality, currently pursuing a Ph.D. degree, is another person

I collaborated with on a daily basis. Next to my thankfulness for your contribution to my thesis are my wishes of good luck in your future career.

I have been privileged to be a part of Software Construction Laboratory, which consists not only of gifted researchers, but most importantly – great people to work with. I wish to thank Viorel Preoteasa, Johannes Eriksson and Linda Mannila for many hours of discussions on research, programming and related matters – I can only hope that I influenced your work in at least a small fraction of how you influenced mine.

My work benefitted greatly from meetings and discussions with other researchers at the Department of Information Technologies. For that, among other things, I am grateful to Chang Li, Torbjörn Lundkvist, Qaisar Malik, Dorina Marghescu, Luka Milovanov, Mats Neovius, Ion and Luigia Petre, Vladimir Rogojin, Elena Troubitsyna, Leonidas Tsiopoulos and Marina Waldén.

The organisational side of my work has been arranged by the Graduate School of Turku Centre for Computer Science (TUCS). I would like to thank the members of the TUCS Board for providing me with financial support that allowed me to focus solely on my research. I would also like to appreciate the work of TUCS secretary, Irmeli Laine, for making even the most complex administrative tasks look easy.

I have received invaluable support from the administrative personnel of the Department of Information Technologies, in particular Britt-Marie Villstrand and Christel Donner, who were always there to help in my fights with bureaucracy. I would also like to sincerely thank the Department secretary, Christel Engblom, for being the one-person solution to any and all organisational problems a Ph.D. student could have during his work.

The software described in this thesis was created by two teams of great programmers. I wish to thank Johan Björkman, Rafael Gallart, Lassi Paavolainen, Kalle Pahajoki and Joacim Päivärinne for their contribution and lessons I learnt from working with them.

The final months of my work on the thesis I spent working in the industry. I would like to thank Joonas Lehtinen for founding and leading *Vaadin*, an outstanding company to work in, and Henri Muurimaa for giving me the chance to become a part of a fantastic team.

My life and work in Finland would not be as enjoyable as it is now, if not without a little help from my friends. A number of them I met during

my research work – they contributed both to my career and invaluable memories.

What ultimately resulted in a thesis about practical application of a theoretical software development paradigm, started as an attempt to apply the same paradigm to the development of correct-by-construction database applications. I would like to sincerely thank Damián Soriano for the cooperation and company, as well as his family for making me feel at home during my short, but unforgettable, visit to Argentina.

I would like to thank Linas Laibinis and Anton Tarasyuk for countless hours spent on discussing research, life in Finland and computer games, often while playing basketball. Maryam Kamali, aside from being a great and valuable person, turned out to be a really proficient curler in times my team desperately needed one. The sharp sense of humour of Bogdan Iancu allowed me to stay sane after endless hours of debugging code and writing the thesis. I learnt many valuable lessons from Charmi Panchal, whom I shared my office with during the last months of my stay at the University. I also wish to thank Kati and Pontus Boström, who keep surprising and inspiring me at almost every occasion.

I am honoured to call Yuliya Prokhorova and Sergey Ostroumov my friends. You bring to my life more, than you can imagine – I hope you know how grateful I am for that.

My stay in Finland would be significantly more difficult, if Mariola, Grzegorz and Weronika Mazerscy were not around together with Andrzej Mizera and Ilona Kruk. I would like to thank you for your support and friendship I received throughout the years.

I wish I somehow could express my endless gratitude to my family. My dear mother, Grażyna, never doubted in me and kept supporting me in every decision I made. Your never ending supply of good advice keeps me going – thank you. My sister, Ola, and her husband, Grzesiek, were always there for me and I am confident it will be this way for years to come. I also wish to thank my father, Maciej, for the support I received during the past years.

My wonderful daughter, Mysia, receives my utmost thanks for reminding me that the most important things in life are to discover joy in everything and to continuously become amazed by simple things.

Last but not least, I wish to thank my beloved wife, Marta, *per l'amor che move il sole e l'altre stelle*.

Table of Contents

Introduction

Computer software is among the most complex products ever constructed by humans. Furthermore, it has always been time and resource consuming. Nowadays, as the role of software is constantly increasing and more and more areas of everyday life are controlled, supported and organised by computers, the issue of quality of software and ease of development is of extreme importance.

Background

There are various approaches to software construction, one of the most important of them today being *object-oriented programming,* which models software system as a collection of interacting objects. Object-orientation is currently supported by the majority of commonly used programming languages.

There are many software systems that are of insufficient quality. Object-oriented programming languages alone do not provide enough guidelines for how to structure the software. In order to construct software of reasonable quality, the programmer needs to combine proficiency in the programming language with knowledge of basic rules and principles of software design. This engineering knowledge, based on years of experience in the domain of software construction, is known as *object-oriented design*.

Stepwise Feature Introduction (SFI) is a software construction paradigm, in which the functionality of software is extended gradually, one feature after another [8]. The construction of a large system is thus divided into a number of smaller, more manageable steps. The resulting software has a layered architecture.

The paradigm focuses not only on the incremental construction of the system, but also strongly advocates its correctness. Stepwise Feature Introduction has been proposed on partly theoretical grounds and has not earlier been applied to the production of large and complex real-life software systems.

The requirements of a software system change frequently during development and also after the release of the software to the customer. Stepwise Feature Introduction takes this into account and includes customer interaction an integral part of the paradigm, providing valuable

feedback for the developers and determining which features are to be implemented and in what order. The paradigm does not specify exact means of achieving and supporting the communication, nonetheless it recommends using some kind of *agile software development process*.

Agile methods in general rely on extensive and frequent communication between the developers and the customers or end users. They also provide guidelines for how to organize the work and how to divide the responsibilities between developers, customers and other stakeholders [16], which is not précised by the paradigm itself.

Problem statement

Stepwise Feature Introduction is a high-level theoretical paradigm for the systematic construction of software systems. As it is common for this kind of concepts, time is needed to evolve from fully theoretical grounds into a commonly used tool. Typically, this time is from 15 to 20 years, and can be divided into six phases [141]:

1. *Basic research*. The basic ideas and concepts are investigated and the initial structure of the problem is formulated.
2. *Concept formation*. A research community is formed around the approach; solutions to specific problems are found and published.
3. *Development and extension*. The idea behind the technology is clarified and an attempt to generalise it is made. The last of the research phases.
4. *Internal enhancement and exploration*. The technology is extended to other domains and used to solve real-life problems; it is also stable so that training material can be developed.
5. *External enhancement and exploration*. A broad community uses the technology, and there is substantial evidence of value and applicability.
6. *Popularisation*. Production-quality versions of the technology are developed.

Currently, SFI can be seen as situated between the second and the third phase. While the basic concepts of the paradigm are stated, solutions to a number of specific problems are still needed.

One such problem, addressed in this thesis, is the application of the paradigm to the development of large-scale software systems. More specifically, we are interested in finding out not only whether the paradigm

is suitable for this task, but also what changes to Stepwise Feature Introduction are needed in order for it to be useful in such settings.

Contribution

The thesis tries to advance methods for software development by combining well-established methodologies for software design and construction with Stepwise Feature Introduction. We have applied the paradigm to the development of two software projects: *ReThink* and *BioImageXD2*. The former is a multiplayer game targeted for a number of distinct hardware platforms, whereas the latter is a highly specialised image-processing software system. These projects differ in size, complexity and serve different purposes. Thus, we can validate our approach, to some degree, against diverse requirements and different environments.

A number of issues related to the application of SFI to software construction in practice was raised during our research. Addressing these problems resulted in combining the paradigm, agile development philosophy and the best practices of object-oriented design and programming. We merged these concepts in order to create a development framework that preserves their best characteristics and is suitable for development of software systems that vary in size and complexity. In particular, Stepwise Feature Introduction provides an organised, layered architecture for the resulting system. Object-orientation brings the ease of development of modern programming languages, as well as the engineering knowledge encapsulated in design patterns and design principles. Finally, agile methods enable quick response to changes in requirements and facilitate an organised development process.

Our work clearly indicates that the paradigm, together with the modifications presented in this thesis, forms a development framework that is scalable and well suited for the construction of large and complex software systems. The main extensions and adaptations of the SFI paradigm that we have worked out concern the layered execution of the software, the specific agile development process chosen, the methods for ensuring quality by testing, and the validation of the quality of the software system using software metrics.

Structure of research

The principles of empirical research [79] have had the strongest influence on the way our research was structured, as presented in Figure 1. Empirical research recommends that the research results are presented in the context and the design of the research experiment, together with its conduct and data collection. Moreover, the gathered data should be analysed and interpreted.

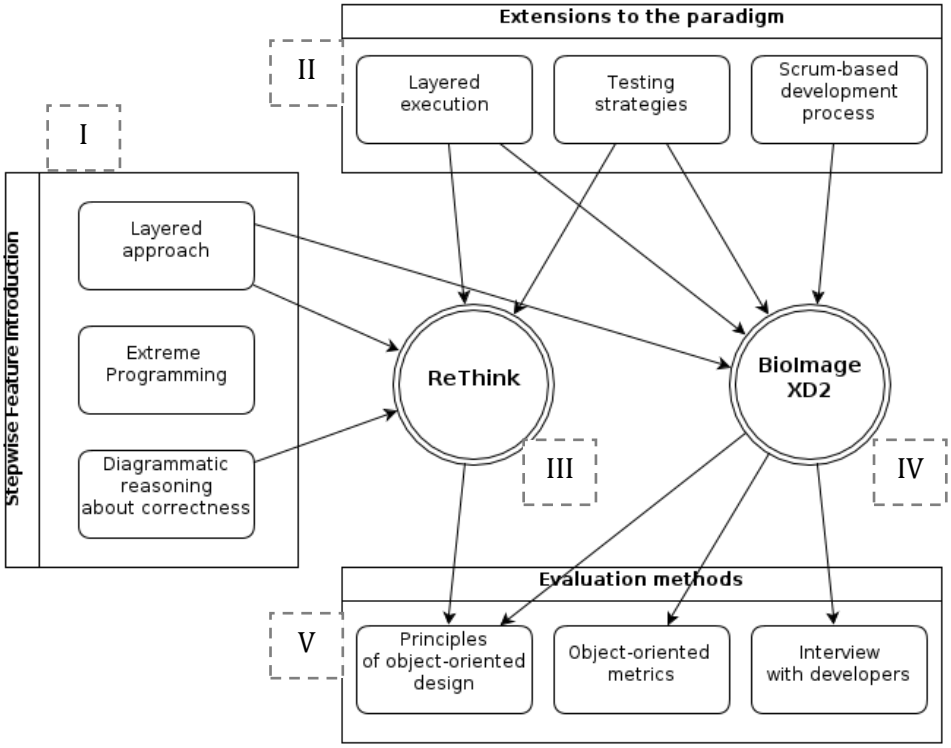


Figure 1. Research overview.

In Part I we present the theoretical background and introduction to the thesis, the paradigm of Stepwise Feature Introduction. The fundamental concepts of the paradigm are presented first. Next we discuss SFI with respect to the characteristics of object-oriented programming languages and agile development, in particular Extreme Programming.

The experiment in our research consisted of performing two case studies [151]. These projects were connected one to another due to the principles of action research, the purpose of which is to influence or change some aspect of whatever is the focus of the research [151][147]. Part II describes the context of our research, the scaling up of the paradigm. More

precisely, we present the extensions that are necessary to the paradigm, so that it fits the needs of large-scale development. We also summarise the objectives and the research questions both case studies are expected to answer [151][136].

Our first non-trivial case study, an interactive board game *ReThink*, is presented in Part III. This software is used as an illustration to the basic concepts of the paradigm. We also identify key elements of architecture of a system built with SFI and outline a diagram-based strategy to ensuring correctness of the produced system.

In part IV, we look at Stepwise Feature Introduction in the large. We describe a complex software system for image processing and analysis called *BioImageXD2*. This software was built with SFI, essentially re-engineering a previous implementation, and required a substantial amount of person years and a large code base to complete. The system and its architecture are described first, after which we present the characteristics of its development process.

Part V of the thesis concerns the evaluation of the paradigm. We confront the paradigm with generally accepted criteria and indicators of good design and high quality. We then compare the results of a measurement plan carried out for *BioImageXD2* to those criteria and generally accepted industry standards.

The final part of the thesis is dedicated to discussion of our research results. We present a number of approaches to software construction and their relation to the paradigm of Stepwise Feature Introduction. The thesis concludes with an overview of our research, threats to validity of our findings and the directions for future work.

Part I:
Stepwise Feature Introduction

1. Fundamental concepts

Stepwise Feature Introduction (SFI) is a paradigm developed by Ralph-Johan Back that allows building the software incrementally, in a layered manner [8]. It is an approach to software construction based on incremental extensions of the system with new features, one at a time.

Introducing new features may possibly alter previously existing functionality, so the paradigm requires the designer to explicitly check that old features are preserved. After adding each feature the system is executable and can be presented to its users and stakeholders, thus gathering important feedback for the next increment. Moreover, each layer, together with the underlying ones, is executable. Thus, in fact a collection of systems is built at the same time.

A software system is seen as a collection of components that interact with each other. We differentiate the components based on the role they play in the system.

Service providers, as the name implies, offer a specific functionality to other components of the system. These providers implement all their functions independently, without relying on or using the remaining parts of the software.

The functionality delivered by the providers is utilised by *service users*. The responsibility of those components is to enable the service to be effectively used during the operation of the system. They are the opposite of the providers in terms of dependencies, as no other components depend on them.

The most common case, however, is that a component in the system provides some functionality by relying on other component or components. Thus, it is both a service provider and a service user – the role depends on the perspective the component is examined from. An illustration of the roles is shown in Figure 2. The service users depend on service providers; however the changes propagate in the reverse order, from the providers to the users.

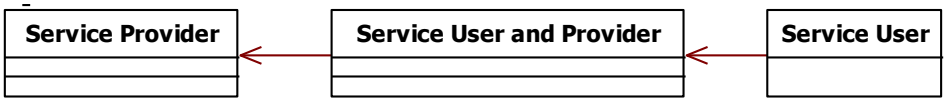


Figure 2. Component roles in SFI.

1.1. Layered structure

When a new feature is added to a service provider, it may require making changes to its users, to be able to use the new functionality. This gives rise to a layered structure, where each layers encapsulates one new feature. Figure 3 shows an example how an existing feature (shown at the top) is extended to provide more advanced functionality (beneath). When introduced, SFI proposed to draw newly added layers on top of the existing ones [8], a practice common in hardware design. However, in object-oriented design there is an opposite preference. This thesis focuses more on the principles of software design; therefore the diagrams present the newly added layers at the bottom of a figure.

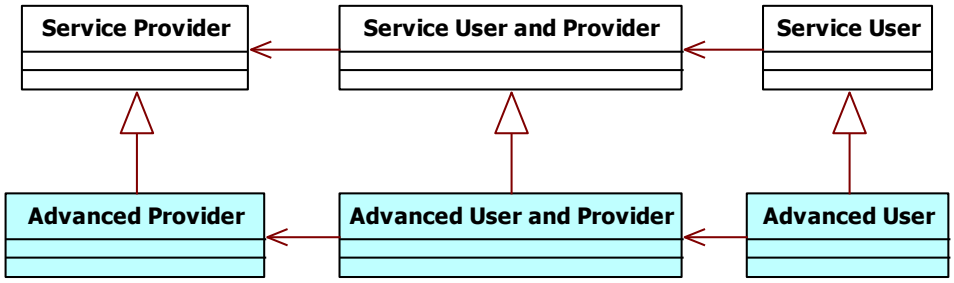


Figure 3. Layered structure of a system built with SFI.

New layer may utilise not only a layer that directly precedes it, but may depend any of the layers introduced earlier, as shown in Figure 4. The structure of layers is not fixed and may be altered through refactoring. This enables not only adding new layers between existing ones, but also replacing existing features with new, possibly more efficient, implementations. Moreover, the internal design of the layers may be changed, provided that the layers directly affected by such modification are updated accordingly.

The changes to the system, in particular the refactoring of existing layers, must result in a new system that is still layered, in the sense that each extended layer, together with basic layers added earlier, forms a fully executable version of the whole system. Such approach gives rise to the collection of systems, rather than a single system, being developed.

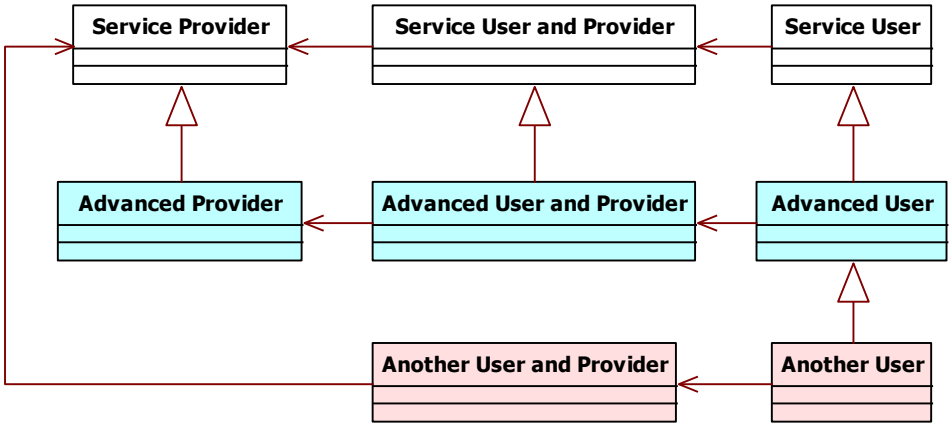


Figure 4. Adding new service user and provider to SFI.

A layer, together with its preceding layers, forms a subsystem, which realises a subset of the overall functionality. This structure is shown in Figure 5, in which the final system contains three layers. Each of the two subsystems and the final system can be reused in a different setting and be extended with functionality different from the original one.

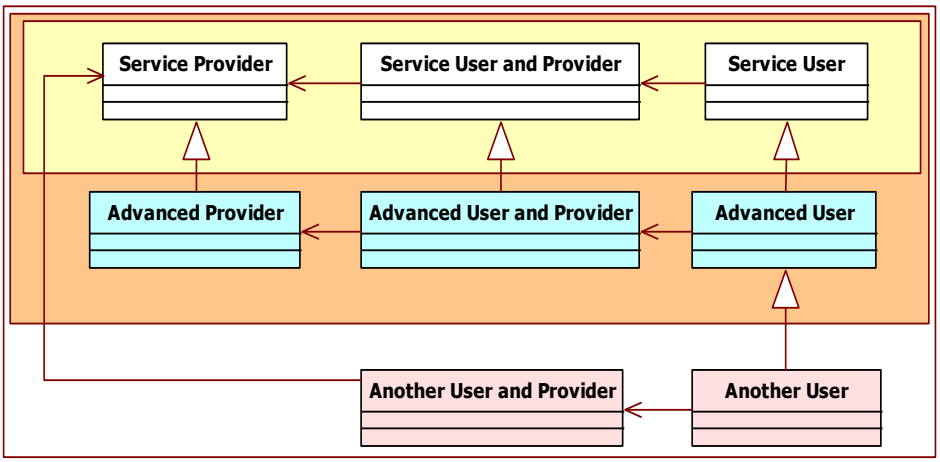


Figure 5. Subsystems in SFI.

1.2. Working with features

The iterative nature of SFI means that the system constantly evolves during its construction. Features may be added, modified or removed as the development continues. Due to a layered structure of the constructed system these changes rarely affect a single component – more often the whole layer containing it, together with the ones depending on it, must be modified.

Service providers can introduce functionality to the system in one of three ways, as shown in Figure 6:

- A new service provider may *extend* the service of an existing provider, by adding more behaviour.
- A new service provider is *added* if it encapsulates required functionality without relying on other providers.
- Lastly, the new service provider may *combine* features of two or more existing providers in order to introduce new functionality.

It is important to notice that the service users do not necessary have to follow this pattern. In fact, subsequent service users are almost solely extending or combining the existing ones in order to support execution of all the features of the system.

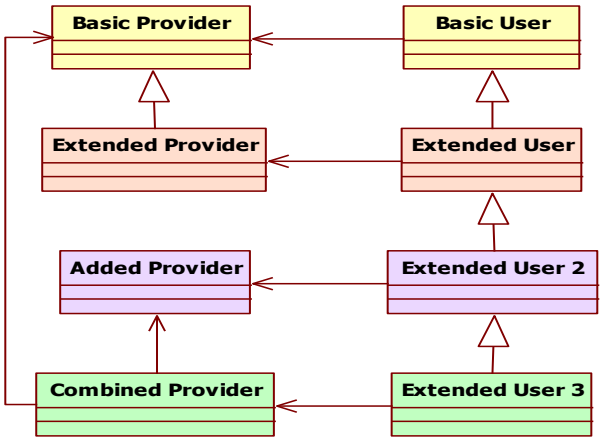


Figure 6. Different methods of introducing features.

We have previously said that the organisation of the layers can be altered to allow more efficient implementation or better design. The layers may be *rearranged*, meaning that the order in which they appear in the system is changed. By r*eplacing* a layer we substitute it with a collection of service users and providers that preserves the original functionality. *Merging* causes two or more layers to be combined into or replaced by a single one. Finally, layers may be *removed* from the system if the functionality they provide is no longer used or needed.

1.3. Elements of system built with SFI

During our work we found the notion of a feature to have different meanings for different groups of people involved in a software project. A customer, a software architect, a project manager and a programmer all

have a different understanding of that term. To avoid the confusion, the paradigm of SFI distinguishes requirements, components, layers and classes.

According to the Cambridge Dictionary, *feature* is a typical quality or an important part of something [35]. The perception of customers follows this common understanding, as they see *requirements* as distinctive characteristics of a software system: something that the system does. System architects also apply the same definition, but from a different perspective. A *component* is a part of the system – something that the system is built of. The programmers have the most detailed point of view on what a *class* is – a part of code that delivers well-defined functionality. Lastly, the perception of the designers falls between the ones of programmers and architects. For them a *layer* is a collection of code-level entities that work together in order to provide certain functionality.

The requirements of a system affect and are realised by its components. The constraints placed on the architecture of the system have an effect on the detailed design of system parts and the way the features should be introduced to the code. These relations are presented in Figure 7.

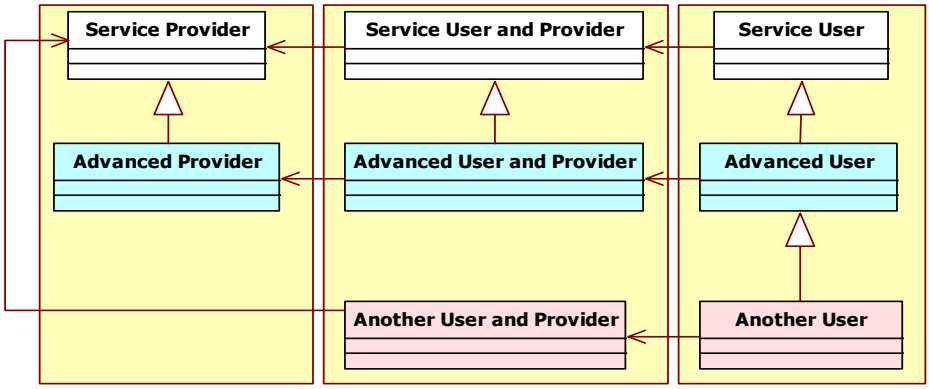


Figure 7. Elements of a system built with SFI.

On the most detailed level we have classes, which contain code that delivers the functionality. The layers and the components are collection of classes. The latter group classes by functionality (shown in the diagram as vertical boxes around the related classes), the former – by dependencies and usage (indicated by the colour of the class boxes). The layers are the basic SFI entity.

1.4. SFI Correctness Conditions

A software system built according to the paradigm of Stepwise Feature Introduction consists of a number of layers. Each of those layers encapsulates a feature, a well-defined increment in functionality. When the initial layer is correct and the introduction of each new functionality preserves the correctness, then, by induction, we can state that the system as a whole is correct.

SFI derives from stepwise refinement [10], which was originally developed for imperative programs [7]. In this sense, SFI can be seen as an extension of refinement that covers object-oriented programming. The layered approach to software construction makes the correctness conditions easy to identify.

By definition software built with SFI is open-ended, meaning that the constructed system can be extended or modified at any time. In order for a feature to provide a solid foundation, on which such future extensions can be built, it is essential to define what it means for a feature to be *correct*. There are four correctness conditions, all of which must be satisfied [8]:

i. *Internal Consistency* – class invariant of the class that implements the feature must be preserved.
ii. *Respect* – the feature must adhere to the constraints of other features it uses; in other words calls to methods of other classes must respect preconditions of these methods.
iii. *Preserving Old Features* – a feature must not break nor alter the behaviour of already existing ones; in particular a subclass must preserve the essential behaviour of its superclass.
iv. *Satisfying Requirements* – all the functional and behavioural requirements set for the feature must be fulfilled.

1.5. Diagrammatic reasoning

The paradigm of Stepwise Feature Introduction proposed reasoning about correctness based on diagrams [8]. The Unified Modelling Language (UML) is created and maintained by the Object Management Group. It provides a language for modelling not only application structure, behaviour and architecture, but also business processes and data structures [123]. UML is a notation intended to be used in an object-oriented design process. There is not much benefit in applying it to other paradigms [73], as it has been

specifically designed to be compatible with the object-oriented software development methods.

Class diagrams are the most commonly used UML diagrams [60]. They present the attributes and operations of classes together with the relations between the classes [123]. Class diagrams are a common form of representing the static structure of the system [52].

In order to accommodate correctness concerns SFI allows several constructs of a UML diagram to be optionally annotated with a question or an exclamation mark. The lack of such symbol indicates that correctness has not yet been considered for a given entity. A question mark signifies that the correctness conditions are not known to hold, whereas an exclamation mark means that the associated conditions have been established. The meaning behind the exclamation mark depends on a construct it annotates, as summarised in Table 1. We describe diagrammatic reasoning based on an example from one of our case studies in the later part of the thesis.

UML construct	Correctness condition
Class box	Internal Consistency, Satisfying Requirements
Association arrow	Respect
Inheritance arrow	Preserving Old Features.
Subsystem box, diagram	Correctness of the subsystem or system

Table 1. Meaning of correctness annotations depending on UML construct.

1.6. Correctness conditions and tests

Constructing software is a difficult and error-prone activity [11]. The complexity of computer programs is greater than any other man-made entity [27]. Furthermore, software is present in nearly all areas of life and frequently controls safety-critical systems. Therefore, there is a need for software not only to deliver the required functionality, but also perform its tasks without errors and faults. In other words we need to prove that the software conforms to the requirements.

The correctness conditions, imposed by SFI, may be satisfied in different ways, depending on the type of the developed system. In most of the cases the correctness is ensured through testing, although other approaches are also possible and sometimes more suitable. Whether or not the system satisfies the initial requirements and delivers the needed functionality depends on the quality and the coverage of the tests. Each of the correctness conditions we presented previously can be ensured by a test of a specific type, as shown in Table 2. The strategy to testing SFI

software is one of the contributions of the thesis and therefore is presented in more details later.

Correctness Condition	Test type
Internal Consistency	Unit
Respect	Integration
Preserving Old Features	Regression
Satisfying Requirements	Unit, Acceptance, System

Table 2. Feature correctness conditions and test types.

2. SFI and object-oriented programming languages

A *programming paradigm* is a set of concepts and abstractions used to represent the elements of a computer program and how the program determines a computation. *Object-oriented programming* is a paradigm in which a system is constructed and executed with objects. All of the actions in such programs are caused by objects sending messages to each other, in the form of indirect procedure calls [166].

The set of key characteristics of object-orientation has not been precisely defined. The term itself is attributed to A. Kay [139], for whom it represents "*only messaging, local retention and protection and hiding of state-process and extreme late-binding of all things*". However, discussions on this topic do more to reveal the prejudices of the participants than to uncover any objective truth about the core matter [137]. Despite this lack of agreement in the community, Simula-67 [122] is generally accepted as the first programming language that utilises the concepts of object-orientation.

We agree with the features of object-orientation proposed by a comprehensive literature survey [6]. The fundamental characteristics are identified based on their occurrence in a variety of sources, including journals, books and conference proceedings. To emphasise the importance of the features the survey names them as *quarks*, as an analogy with the fundamental constituents of matter [53]. Moreover, these concepts are divided into two groups [6], as shown in Figure 8.

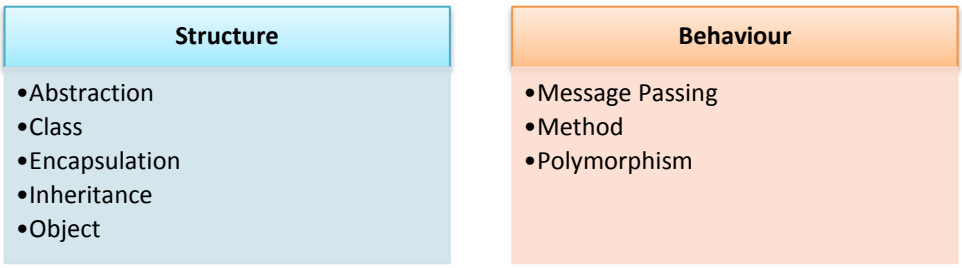


Figure 8. Eight quarks of object-orientation.

The paradigm of Stepwise Feature Introduction does not place any constraints on the programming language, although it indirectly requires an object-oriented one. We distinguished three basic operations that can be performed on features: adding, extending and combining. Two possible roles of each software component in a system built according to the SFI paradigm were also identified: service providers contain the functionality and service users utilise it or make it available to the users of the system. Each of the operations results in a new layer with service provider and service user being added to the system.

The programming language used with SFI must support at least inheritance and subtype polymorphism – or similar mechanisms – to realise extending of the features. In fact, a typing system and inheritance both have an effect on how each of the operations is realised.

2.1. Subtype polymorphism

The concept of *polymorphism* – understood as the ability to hide different implementations behind a common interface [196] – was used in software also prior to the introduction of object-oriented programming. In the context of object-orientation, polymorphism can be shortly defined as the ability of different classes to respond to the same message, each implementing the method appropriately [6].

Object-oriented programming languages provide polymorphism in different ways. In *statically typed languages* the type of an object is specified in the source code and type checking occurs usually at compilation time for the whole code. Thus, it is possible to check whether an object can be used in a given context before the code is executed, typically by examining the type hierarchy. This approach is mostly used in compiled languages.

Dynamic typing is when the type of an object is examined at run time. As opposed to static typing, the execution flow must reach a part of

code for it to be type-checked. This approach significantly increases the importance of unit tests, as they can help in detecting and fixing run-time type errors at development time. Languages with dynamic typing are usually not fully compiled but either interpreted or compiled to byte code.

A variant of the dynamic typing, called *duck typing*, has emerged recently and is applied in modern scripting, object-oriented languages. The name is derived from J. W. Riley's *duck test,* usually phrased as "*When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck*" [47]. Duck typing is concerned with the behavioural aspects of an object rather than its type. More specifically, the possibility of an object to deliver required functionality is checked, not its place in a particular type hierarchy.

The paradigm of SFI utilises subtype polymorphism when new features are introduced to the system. Polymorphism ensures that these new features can be used in the context of the old ones, although it does not guarantee that the new code preserves the functionality of the old one. Polymorphism also enables one or more features to be replaced, e.g. by their more efficient implementation.

Languages with static and dynamic typing place similar requirements on service providers that extend existing functionality. More precisely, it is required that the extending provider is a subtype of the extended provider. This condition is needed to ensure that the new functionality can be used in the same contexts as the existing one. Introducing new features to the system, whether by addition or by combination, is, however, handled regardless of the programming language typing system. In both cases a new type should be created; in the latter case type composition is used to combine features.

In statically or dynamically typed programming languages the users of added, combined and extended features must belong to the same type hierarchy as the users in the previous layer. This allows the new classes to be used in place of the old ones.

Languages with duck typing do not require service users or providers to belong to any type hierarchy. Instead, the class must be able to respond to the calls proper for the class from a previous layer and deliver the same results. In particular, this is required from the service providers that extend existing functionality and from the users of added, extended or combined features.

2.2. Inheritance

First introduced in Simula-67, *inheritance* has been suggested as the only unique contribution of the object-oriented approach to system development [69]. It is a mechanism that allows the data and the behaviour of one class to be included in or used as the basis for another class [6].

Inheritance is also known as subclassing – it allows new classes to be derived from old ones by adding implementations for new methods and, when necessary, selectively overriding implementations of the old methods [137].

There are a number of types of inheritance supported by different programming languages. We distinguish four major implementations of this characteristic: single, single with multiple interfaces, single with multiple mixins and multiple.

The use of inheritance in Stepwise Feature Introduction is rather straightforward. Extending the existing features and adding new ones to the system, if not impossible, would be tedious and difficult without inheritance. Moreover, inheritance ensures that the parts of the features not modified or extended in a new layer maintain their original behaviour.

Combining existing features is an operation that is affected the most by the type of inheritance supported by a programming language. Multiple inheritance allows to replace combination with extension; in such case a new feature extends a number of existing features at the same time. Single inheritance with multiple mixins, on the other hand, provides a simple mechanism for combining features by including their code in a single class. In most of the cases, however, composition should be favoured over inheritance [61]. Object composition is also the only possibility to combine features using single inheritance or single inheritance with multiple interfaces.

The paradigm of SFI requires a form of inheritance from a programming language. The type of it, however, is irrelevant, as the operations on features can be performed regardless of it Therefore, the choice of type of inheritance can be done indirectly, as part of selecting a programming language to be used throughout the development.

3.SFI and Extreme Programming

Applying Stepwise Feature Introduction to software development provides a systematic approach to the construction process and enables project to be evaluated after each development step. The latter implies that the management of the project resources becomes more effective.

Stepwise Feature Introduction does not enforce using a particular software process; however processes bearing similar characteristics are rather obvious choice due to their iterative and repeatable nature. Thus, by definition, Stepwise Feature Introduction is suitable for all iterative, adaptive and continuous development processes known under the general name of *agile development processes*.

Extreme Programming (XP) is an agile software development methodology [14] focused on frequent releases of the product code in short development cycles. XP also emphasises the incremental construction of software and participation of the customer in the planning and development process. The method derives its name from taking the beneficial practices of traditional development to the extreme. XP can be also seen as a discipline for organising people rather than as a pure development methodology, as it puts heavy requirements on how and in what order certain activities should be performed. Moreover, the goal of the Extreme Programming is to manage the development team to effectively produce software of high quality [14].

XP, as other agile development methods, relies on releasing functional versions of the final product in small, repeatable cycles. During each cycle the method distinguishes four basic activities to be performed: coding, testing, listening and designing. Apart from these activities XP identifies a number of values [177] and a set of practices [178]. The stakeholders in the development process are the development team, jointly responsible for implementing a working product, and the customer, who is in charge of setting the requirements of the software being built.

The structure of the software built according to the Extreme Programming method is not clear, as the methodology itself de-emphasises documentation or careful design, and suggests postponing architectural changes until they cannot be avoided. The lack of overall structure of software built with Extreme Programming has been a subject of criticism [163] of this method.

SFI does fit quite well with Extreme Programming [14]. The paradigm of SFI can be seen as a complement to the process, as it provides means to structure the software and build its components in an incremental manner [8]. This approach has been successfully applied to a number of projects carried out at Åbo Akademi University [9].

3.1. Activities

Coding is considered to be the most important part of the entire development: without the code it produces there is no working product. Extreme Programming advocates *pair programming*, a technique in which two programmers – named *driver* and *navigator* [194] – share one workstation to deliver the code. The driver has the control over the keyboard and other input devices and is responsible for typing the code. The navigator, on the other hand, reviews and suggests improvements to the code as it is being typed, as well as is responsible for strategic thinking about the direction of the work. Whenever needed, the two programmers can brainstorm any obstacles they identify during the process. The roles of driver and navigator are switched frequently to ensure equal productivity of both programmers.

Test-Driven Development [15] is often applied together with Extreme Programming. Unit tests are required for every piece of code that has been implemented. The testing conditions are written not only by the programmers responsible for the code being tested, but also by other members of the development team. Acceptance tests [14] are also carried out in order to verify that the implementation provided by the development team matches the expectations of the customer (or the final users).

Listening is the only basic activity of XP that does not refer directly to implementation. The purpose of this action is for the developers to understand what are the customer requirements, what is the business logic of the software and what design decisions must be made. Moreover, this activity enforces a communication with the customer, who must be available on-site for the development team while the product is being built.

Designing is the activity that complements the remaining three. Without an overhead design the development team would likely reach the point in which extending the software is more expensive than building its new version completely from scratch. Therefore, the programmers – more precisely, those holding the role of navigator in pair programming – are responsible for designing and implementing a proper structure for the code.

3.2. Values

The key value of XP is the *simplicity*. It is understood as doing precisely what is currently required and nothing more. Moreover, simplicity enforces that the work is done in small, simple and easily manageable steps that bring the developers closer to the final product. XP encourages the programmers to implement the simplest working solutions to the requirements set at any given moment and to avoid designing for possible future changes and extensions.

Communication, as a value, deals with the inter-human relations of the development team and the customer. Face-to-face daily interaction is essential to understand the requirements and implement them. Moreover, communication helps establishing a *team spirit*. The development team works together to achieve a goal common to all its members.

Feedback is an essential part of all agile development methods. Short iterations, during which a working version of a product is delivered, allow the customer and other stakeholders to provide comments and directions frequently. Extreme Programming establishes a number of planning and feedback loops that repeat during the development, as seen in Figure 9 [179].

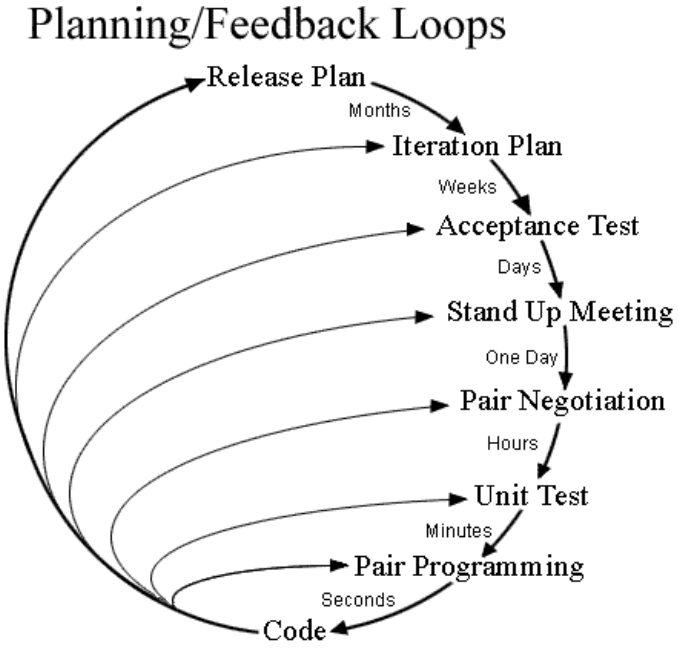


Figure 9. Planning and feedback loops in Extreme Programming.

Respect is a crucial value for enabling good communication and feedback. Each of the team members contributes value to the project, therefore must receive and feel the respect they deserve. Furthermore, the developers are required to respect the expertise of the customers and vice versa. This value also directly imposes the responsibility of the team for the delivered implementation and of the customer for the requirements set.

Finally, the team and the customers should have *courage*. It is required to tell the truth about the progress and estimates regarding the development. Moreover, the customer is obliged to provide honest feedback to the team and should not be afraid to modify the requirements, even if some of them are already implemented. The team, on the other hand, should have the courage to work together and to adapt to the changing requirements.

3.3. Practices

The practices – also referred to as rules – of Extreme Programming are divided into five groups, each concerning one area of the development process: planning, managing, designing, coding and testing. We summarise these practices in Table 3 [178].

Area	Rule
Planning	User stories are written.
	Release planning creates the release schedule.
	Make frequent small releases.
	The project is divided into iterations.
	Iteration planning starts each iteration.
Managing	Give the team a dedicated open work space.
	Set a sustainable pace.
	A stand-up meeting starts each day.
	The Project Velocity is measured.
	Move people around.
	Fix XP when it breaks.
Designing	Simplicity.
	Choose a system metaphor.
	Use CRC cards for design sessions.
	Create spike solutions to reduce risk.
	No functionality is added early.
	Refactor whenever and wherever possible.

Coding	The customer is always available.
	Code must be written to agreed standards.
	Code the unit test first.
	All production code is pair programmed.
	Only one pair integrates code at a time.
	Integrate often.
	Set up a dedicated integration computer.
	Use collective ownership.
Testing	All code must have unit tests.
	All code must pass all unit tests before it can be released.
	When a bug is found, tests are created.
	Acceptance tests are run often and the score is published.

Table 3. The rules of Extreme Programming.

Part II:
Scaling up SFI

4. Design challenges

Constructing software is a complex and demanding task. When building a large-scale software system, the task becomes even harder. In addition to a high number of requirements, there are additional considerations to be taken into account.

Whether or not the paradigm of Stepwise Feature Introduction can be applied to construction of large-scale software systems had not been previously examined. The suitability of SFI in the context of developing such systems is thus the primary scope of our research.

However, our work is not solely focused on evaluating whether or not the paradigm can be of help when constructing large-scale software. Rather, we aim at identifying potential problems such integration could raise and modifying the paradigm to address them.

4.1. Potential problems with scaling up

Based on our experience we can anticipate major issues caused by the scale of the developed software. Software that is complex and large usually requires an architecture, in order for it to be maintainable. Furthermore, a development process must be organised to provide developers a possibility to construct the system in a reasonable way, according to the specification. Finally, the quality of the software must be high – or at least it needs to reliably perform its tasks without failures.

In the context of our work the most important consideration with large software system is the fact that the larger the software system is, the more likely it is to have a defined architecture. By that we understand a clear high-level design in the form of a collection of interconnected abstract entities, with defined roles and responsibilities. This kind of system design is usually created upfront, as soon as initial specification or requirements are known, and later modified to accommodate changes.

The development of large and complex software systems requires significant resources. To effectively write code and deliver a final version of the system a development process is needed. It divides the work among involved people and defines means of interaction between them. An inefficient process may lead not only to delays in delivering the system, but also in constructing a software that does not fulfil its requirements or meet the set standards.

Finally, the quality of complex systems is essential, as the costs of malfunctioning large system are usually in proportion to its complexity and development cost. The key quality attributes should be decided as soon as possible, and the system should be monitored constantly during the development to ensure that it possesses all needed characteristics.

The issues affect one another. The architecture depends, among other things, on what are the desired quality attributes of the system, e.g. its maintainability or reusability. In turn, the development process has to follow the design of the system and allow the development team to construct the software. Lastly, the quality assurance is built into the development process, so that it is possible to measure the quality attributes during the development and react in time.

4.2. Defining case studies

To confirm our anticipations about problematic areas of the paradigm and modify it to support development of large-scale software, we decided to conduct two case studies. Both of them are software development projects, although with different goals and complexity. Additionally, they were performed in sequence, so that the findings of the case study performed first can be applied to the other.

Pilot case study

Careful planning is required in order for a project to be a meaningful case study. We expect from our pilot case study to identify those aspects of SFI that may be problematic when the paradigm is applied to a construction of large-scale software system. Consequently, we aim for mediocre complexity of our first project. There are several issues that need to be addressed before the project starts [151][147]. Their resolution for our pilot case study is summarised in Table 4.

ReThink, which we present in Part III, was a straightforward choice for the first project. The project had many properties of a typical industrial system: stakeholders, deadlines, requirements, and the like. At the same time it was developed in a controlled, academic setting that allowed conducting research and having control over e.g. the development process.

Issue	Question	Solution
Objective	What to achieve?	Identify areas of SFI that need to be modified when applied to a construction of a larger system.
The case	What is studied?	The development process and the design of *ReThink*.
Theory	What is the frame of reference?	The paradigm of SFI and the principles of object-oriented design.
Research questions	What to know?	What areas of SFI are likely to be an issue when scaling the paradigm? Which areas of SFI can be modified without affecting the core principles of the paradigm? How elements of SFI can be mapped to parts of a typical development setting? How to ensure correctness of the constructed system?
Methods	How to collect data?	Analyse the development process and strategies applied to testing and design.
Selection strategy	Where to seek data?	Inspecting system design and source code.

Table 4. Case study plan for pilot case study, *ReThink*.

Large-scale case study

The main purpose of the large-scale case study is to answer those research questions set in the thesis that were not answered by the pilot case study. We have developed *BioImageXD2* during the course of this case study. The project is described in details in Part IV. It was conducted with the use of the modified version of the paradigm, which we present later in this chapter. The plan of the case study is shown in Table 5.

Issue	Question	Solution
Objective	What to achieve?	Evaluate whether or not the modified version of SFI allows constructing high-quality, large-scale software systems.
The case	What is studied?	The development process, the design and (to a lesser extent) the implementation of *BioImageXD2*.
Theory	What is the frame of reference?	The paradigm of SFI, the principles of object-oriented design and the perception of the stakeholders.
Research questions	What to know?	Can SFI be used to ensure that the produced software has high quality? Which areas of SFI do not scale and do not fit development of large software systems?
Methods	How to collect data?	Analyse the development process and strategies applied to testing and design. Conduct interviews with the stakeholders.
Selection strategy	Where to seek data?	Interviewing project stakeholders; inspecting source code and design of *BioImageXD2*.

Table 5. Case study planning for *BioImageXD2*.

4.3. Altering the paradigm of SFI

Stepwise Feature Introduction is an organised approach to incremental software construction. It relies on the two essential principles provided by object-orientation, namely inheritance and subtype polymorphism. The paradigm is also based on small increments in functionality and frequent interaction between the development team and the customer. In this aspect it shares its characteristics with agile development methods; however, such development process is not explicitly required.

The paradigm of SFI addresses – in theory – all of the major issues caused by large development scale. It organises the constructed software in layers, thus providing an architecture upfront. Due to the nature of the paradigm the functionality of the software is divided into small, manageable steps, which are then realised with an iterative development process. Finally, the correctness is preserved from one iteration to another, leading to higher quality of the constructed system.

The lack of other significant requirements allows the paradigm to be customised, depending on the development setting. In other words, it is

possible to use the principles of SFI to organise a custom development process, thus turning SFI into a general, high-level development framework.

Our attempts to scale the paradigm and apply it to development of software of significant size and complexity indicated a number of areas that need to be specified in more details. These areas correspond to the issues addressed by the paradigm and raised as concerns during its scaling up. In other words, we need to provide concrete methods of solving problems in ways that do not contradict the paradigm.

Layered execution

Software constructed with the Stepwise Feature Introduction is built as a collection of layers. Each layer in the system (together with layers below it) is intended to be executable. This executability of each layer must be preserved also when the software in modified and layers are rearranged.

We have established a number of methods that ensure layer executability. These methods are based on the principles of object-oriented design and the properties of Stepwise Feature Introduction. Moreover, they are applicable to any system constructed according to the paradigm, regardless of its size and complexity.

Testing

The issue of software correctness had a significant impact on the way the software systems we present in the thesis were developed. This thesis focuses on application of Stepwise Feature Introduction to large-scale software systems that were developed in an academic environment, with limited resources and schedule. Therefore, correctness was approximated with testing, code reviews and other commonly applied techniques for ensuring quality of software and checking that it conforms to the requirements. The principles of correctness stated by SFI were used to organise and guide the testing processes.

Agile development process

Stepwise Feature Introduction was designed to be used with an agile development process. However, the details of such process were left unspecified. During the development of our case studies we decided to design a process that is dedicated to Stepwise Feature Introduction.

We based our work on Scrum [153], an iterative agile development framework. The representative of the end users is strongly involved in this process, indicating at the end of each iteration (sprint) the direction for

further development. The development team, on the other hand, controls how much functionality will be implemented during each iteration. The contents of a sprint cannot be modified during its duration; thus the development process precisely defines the moments at which changes of the functionality can occur. Scrum shares many of its characteristics with Stepwise Feature Introduction, thus it can be integrated seamlessly.

5.System execution

The system constructed according to the paradigm of SFI must be executable after each added layer. Object-oriented programming languages provide a variety of methods to ensure that this vital requirement is met. The source code may contain a number of entry-points, i.e. points that may start its execution. In order to execute the system, however, a single entry-point must be selected manually.

5.1. Executable method

A method that executes the software from the layer it is implemented in is the most straightforward approach. The simplicity of this solution is its main advantage. Furthermore, it enables a layer to be executed in more than one way by providing more methods, as seen in Figure 10.

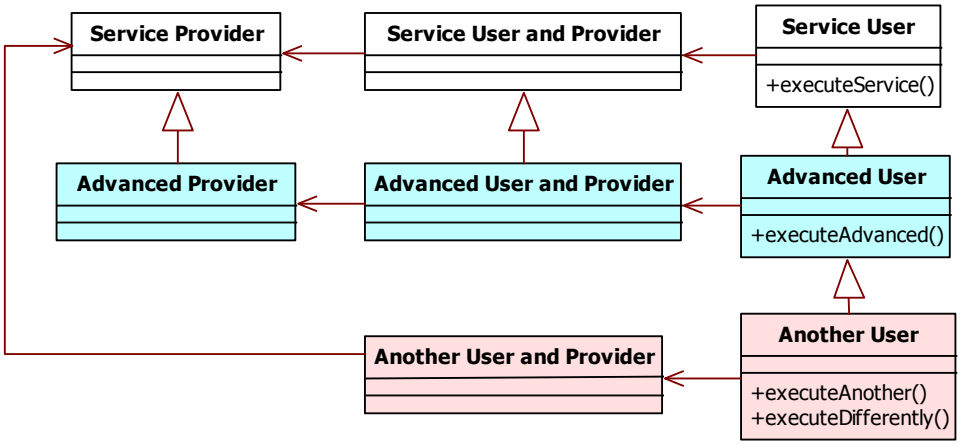


Figure 10. Executable method.

The drawback of this solution is that either the method or the class must be manually specified and called when executing the system. Moreover, the platform-specific implementation details related to execution must also be

included in the method, limiting the reuse and negatively affecting the design principles.

5.2. Inheritable executable method

Object-orientation provides inheritance, which enables a subclass to retain some of the functionality of its parent. The executable method may thus be made an essential characteristic of the type hierarchy. This allows the execution to be fine-tuned in the subsequent layers in case the new behaviour requires such action. The inheritable executable method is presented in Figure 11.

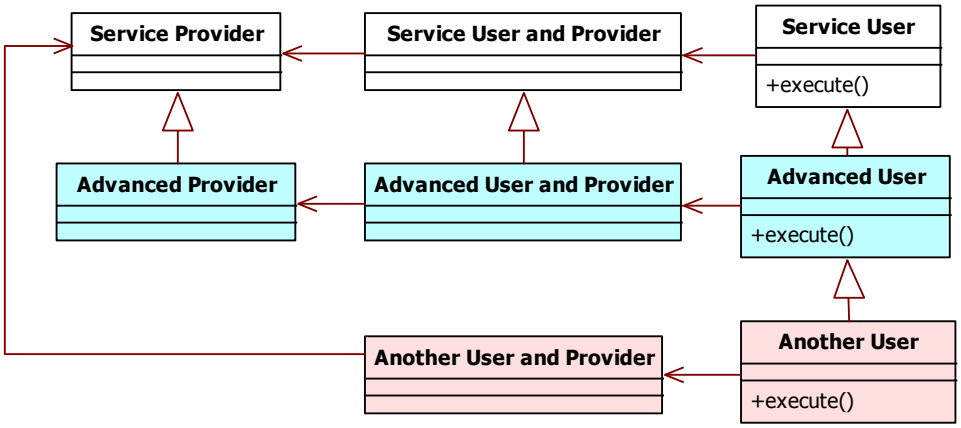


Figure 11. Inheritable executable method.

The main disadvantage of this approach is that inheritance may work with instance methods only, depending on a programming language. In other words, an instance of an object must be created before the method may be called. This introduces additional complexity when executing the software.

5.3. Dedicated service user

The code related to the execution of a layer can be encapsulated in a single class. Such class belongs to the layer, shares its dependencies and provides a well-defined functionality; hence it follows the principles and requirements of both the design and SFI. Furthermore, such class provides a single entry-point to the layer to be executed, regardless of the system the layer is used in. The dedicated service users are shown in Figure 12.

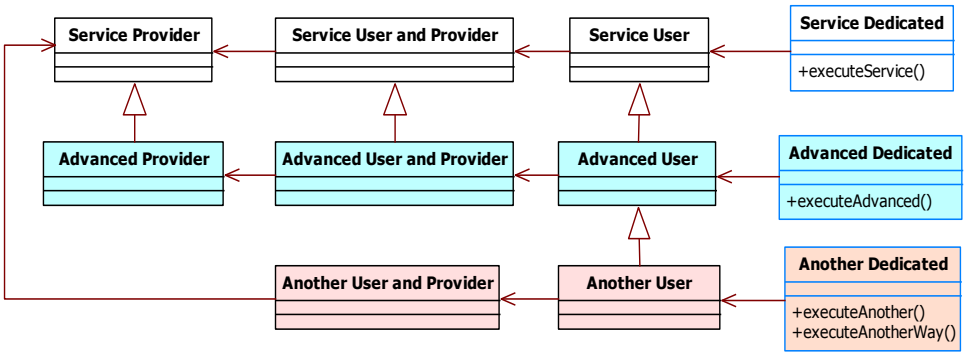


Figure 12. Dedicated service users.

This solution introduces additional classes to the system; hence it increases the overall complexity and the number of dependencies. Furthermore, unit tests are required for these classes. Writing such tests may be a challenge, since these classes are focused on running the software and presenting it to the end-user.

5.4. Hierarchy of dedicated service users

The classes that execute the subsequent layers may inherit one from another. This hierarchy allows its classes to override parts of the execution process whenever needed to include layer-specific behaviour, as presented in Figure 13.

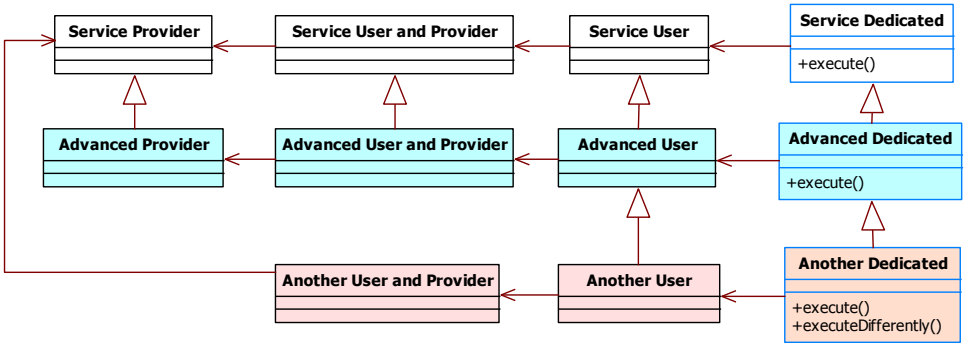


Figure 13. Hierarchy of dedicated service users.

Benefits of this method are similar to those of a stand-alone dedicated service user. Another advantage is that, due to inheritance, parts of functionality related to layer execution can be shared within the hierarchy. The major drawback, however, is the increased complexity of rearranging the layers.

This approach to ensure executability was an optimal solution in one of our projects, *ReThink*. Due to its requirements being fixed already at the beginning of the development, the layers were guaranteed not to be rearranged. Therefore, the hierarchy of dedicated service users could have been established to share code related to execution among different platforms.

5.5. Dedicated system executable

All the methods mentioned above require that a layer to be executed is manually specified by the end user. This reveals the internal details of the system and in most cases should be avoided. However, the layer to execute can be indicated directly in the source code of an external class. This solution may be especially useful when the system is released after each layer.

The stand-alone system executable must be updated every time a new layer has been added to the system. This may be omitted provided that it is possible to perform the execution based on a configuration file, as shown in Figure 14, or other external resource, if the programming language offers such features.

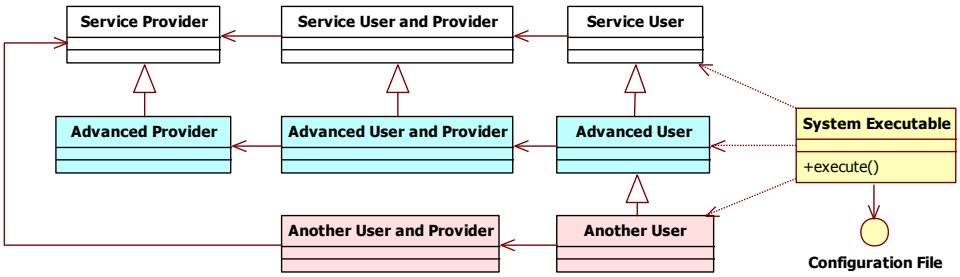


Figure 14. Dedicated system executable.

5.6. Combined approach

In most cases one of the presented methods of layer execution is sufficient and suitable for the system in construction. Large-scale software systems, however, may require a combination of these approaches. The complexity of the system may cause its parts to require a dedicated way of execution. Moreover, different components may be developed independently and thus the layering scheme may not be common for the system as a whole.

An example of such approach can be observed in our large-scale case study, *BioImageXD2*, that deals with image processing. Our solution was based mostly on a dedicated system executable that enabled the

software to be executed at a specified layer. However, due to complexity of *BioImageXD2*, several other enhancements were made. In particular, the executable was modified to allow not only a certain layer to be specified, but also a component or even an individual class to be loaded. Furthermore, the dedicated executable was responsible for handling interaction with the end-user and creating the image-processing pipeline.

The abovementioned solution was combined with a modification of executable methods. The declaration of each module type present in the software contains a method responsible for performing task specific to module type (file loading, file writing, data displaying, data processing). This method was called in the context of the dedicated executable, causing module to perform its task. Due to the construction of image-processing pipeline, however, such method cannot usually be called outside of the dedicated executable.

The main benefit of this combined approach was the ability of the system to be extended with plug-ins without altering the executable. By specifying a single entry point to each module we also limited the dependencies and provided a clear, straightforward mechanism for future extensions. An obvious disadvantage of the solution is its limited reuse, as it had been designed and implemented to fit *BioImageXD2* specifically.

6. Testing software built with SFI

The paradigm of Stepwise Feature Introduction adds the notion of *correctness* to the development, albeit the developers can decide how this issue is handled. For some cases it may suffice to perform rigorous testing or code reviews. On the other hand, in case of formal developments the correctness ought to be mathematically proven.

Contrary to most other paradigms, SFI makes correctness an integral part of the development process. In other words, the software built in accordance with SFI is constructed correct, although with respect to the definition of correctness used in the development. Various techniques can be applied to ensure the correctness of the system, depending on the criticality of the system and the allocated resources.

In typical software development, the goal of correctness is approached with carefully designed test suites. As opposed to formal verification, testing will never prove the absence of errors – nonetheless it

greatly increases the confidence that a program works as expected [127]. The correctness of a software system is thus not proven, it is only approximated.

With SFI the software is built incrementally. Each feature added to the system is encapsulated within a layer that contains a number of classes. Each of these classes falls into the role of service provider or service user, or both. Approximating correctness with testing depends on the role of each class. The way a feature is introduced into the system – whether by adding, combining or extending – also affects the approach to testing, namely by imposing correctness conditions to consider. The general approach to testing, however, does not differ significantly from the one used in testing object-oriented software and can be modified according to the needs of the particular project it is applied to.

6.1. Test-Driven Development

Unit tests verify the functionality of an individual software unit [169], usually at the function level [19][181]. We can benefit from *Test-Driven Development* [15] in the design, implementation, and testing of the software units. A typical development cycle for test-driven development is shown in Figure 15.

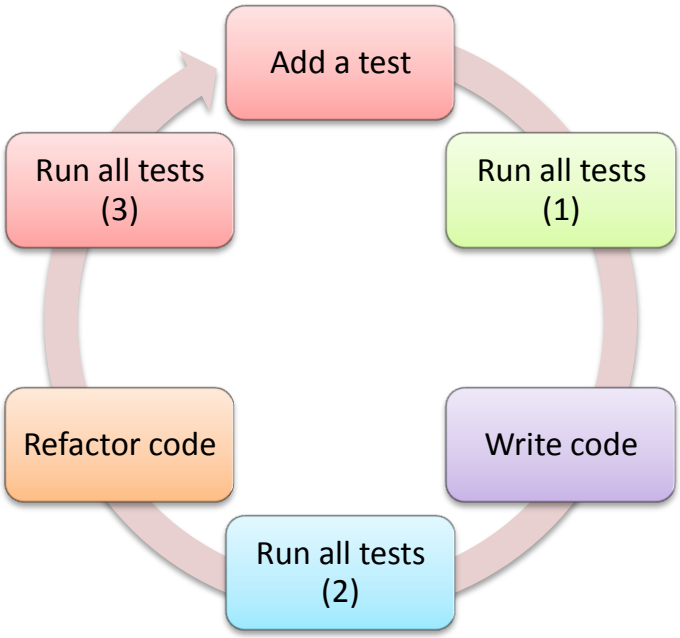


Figure 15. A typical test-driven development cycle.

The cycle starts by writing a test for a new, yet unimplemented, piece of code, either as a separate unit test or as a modification of an already existing one. This approach enables the programmers to focus on the interface of the class and how it can be used. Moreover, the requirements for the class are explicitly stated in the unit test. This constitutes a major difference when compared to the regular development processes that start with implementing the code and testing it afterwards. Test-Driven Development means that the order of these operations is reversed.

Once the unit test is written, the tests for the whole system are executed. Such collection is known as test suite and includes all unit tests for all the classes in the system, as well as other tests of different kinds. The tests written earlier must pass, while the newly added must fail, in order to prevent adding (by mistake) a test that always succeeds. Typically a stub of the implementation, which returns null references or meaningless values, is also written to fulfil this request.

Code that passes the new test is written next. The programmers should not try to deliver a highly optimised solution to the problem; they should focus on providing simple code that passes the test. This ensures that no untested functionality is added to the system.

After the implemented class passes its specific unit test, the test suite must be executed again. All the tests (both old and new) must succeed to guarantee that the new code does not break the existing functionality while achieving the new functionality.

The next step in the cycle focuses on improving the newly added code. The code should be refactored to the point where it meets the quality standards for the project e.g. by removing code duplicates, optimising loops, etc. The development cycle ends with running the test suite and ensuring that all the tests pass after the refactoring.

6.2. Unit tests and service providers

We benefit from unit tests mostly when working with service providers. The correctness condition that has the most influence on such component, regardless of the way it was introduced to the system, is its internal consistency. We utilise unit tests to verify that the code of the service provider is internally consistent and does not violate the constraints. The unit tests are also used to ensure that the newly added code satisfies the customer requirements. Thus, the unit tests are used not only to check

that the code implements the service properly, but also implements the proper feature.

An example of unit test from our case example *ReThink* is given in Listing 2 (Appendix 3). The test checks whether class `RectangleBoard` delivers the desired functionality. An object of the tested class represents a rectangular board that has a specified number of columns and rows. The tests ensure that the functional requirements hold, i.e. the board is initially empty, its dimensions are as requested and that a counter placed on the board can be received. Methods to obtain instances of tested objects and their properties are also defined, so that the tests can be reused by another unit test later.

6.3. Regression tests and service providers

Regression tests focus on finding defects after a major code change has occurred. They prevent unintended consequences of program changes, when the newly developed part of the software interferes with previously existing code [169][181]. The purpose is to ensure that the old functionality is preserved while adding new code.

Performing and organising regression testing is known to be expensive process [149][66], mainly due to fundamental issues it has to address. To minimise the costs, the testing should be executed only on those parts of the system that might have been affected by changes. Such parts should be ideally identified automatically to avoid the tedious process of analysing the dependencies between system components. Regression testing by itself does not specify the method or the strategy with which the affected components should be tested. Additionally, reusing tests between subsequent releases of the software system is also an important issue in the design of regression tests.

These issues have been addressed by various researchers in the past years. A number of techniques to extract the affected parts of the system are available [66][3][149]. Moreover, a comprehensive regression testing process specific for object-oriented software has been established [82]. A strategy for using regression testing with agile software development has also been proposed [113].

The check for preserving the old functionality must be done whenever a service provider extends an already existing one. This is

achieved by applying regression testing to the new code. With object-oriented programming languages it is possible to realise regression testing with carefully designed unit tests. Due to inheritance and subtype polymorphism an extending class can be used in context of the base class, in particular when executing the unit tests. Therefore, the aim of regression testing – ensuring that the old functionality is unchanged – is preserved.

A class `RethinkBoard` from our case study *ReThink* is an example of a service provider that extends a previously existing one, in this case `RectangleBoard`. The code of the unit test for class `RethinkBoard` is shown in Listing 3 (Appendix 3).

The unit test class inherits from the test for its superclass. The methods for accessing instances of tested object are overridden (lines 22-32). This means that the tests for `RectangleBoard` will be executed on an instance of `RethinkBoard`, thus performing regression tests. Two additional tests check whether newly added behaviour works as expected (lines 34-60, 62-88).

The inheritance tree of the production code is reflected in the tests, as shown in Figure 16. The class `RethinkBoardTest` is a subclass of class `RectangleBoardTest` and inherits its methods, in particular the tests. Reflecting the class hierarchy in the structure of the unit tests allows us to perform regression testing at the same time.

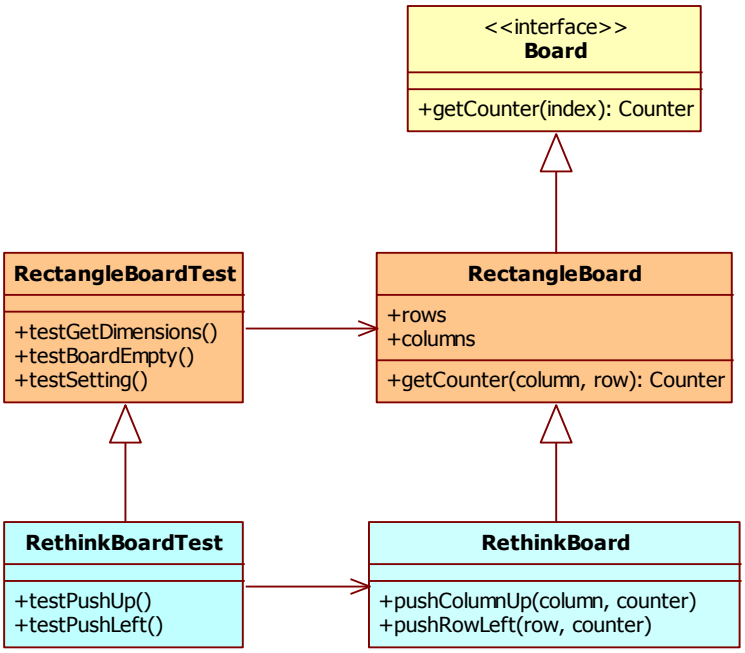


Figure 16. Hierarchy of test cases in *ReThink*.

6.4. Integration tests and service users

Software units are combined and tested in *integration tests* to evaluate the interaction between them [169]. We use such testing to verify the interfaces between components against a software design, as it exposes defects in the interfaces and interaction between modules [17][181].

There are several approaches to integration testing, differing mainly in the order of adding modules to the system. In *bottom-up integration testing* the concrete modules are developed and tested first. The more abstract modules are added and tested gradually next, until the complete system is assembled and tested.

Top-down integration testing is the opposite. The testing starts with the abstract modules; the concrete modules are progressively added one by one. The functionality of the abstract modules is replaced with test stubs during early testing, to mimic the desired behaviour. The stubs are changed to the original modules as the tests progress.

In the *Big Bang approach* to integration testing the modules are not integrated before the system (or a significant part of it) is complete. Checking connections between software modules is thus postponed to a later stage. This method is considered useful only for small software systems and should be avoided in more complex development [17][84].

Variants and combinations of the above methods have also been proposed by the community [117][88][84]. However, the actual strategy of integration testing should depend on the specific project it is applied to.

The crucial part of each service user is not to violate the constraints of the service it utilises. Hence, this is the most important correctness condition that needs to be tackled. Integration testing provides a way of ensuring that different components of the system are able to communicate with each other and that the system (or its currently tested part) works properly as a whole. Therefore, they are a suitable way for testing whether service users respect the constraints of service providers.

Stepwise Feature Introduction is an approach to bottom-up system development. The system starts as a single layer with concrete, simple functionality that is later extended in subsequent layers. The bottom-up approach is thus recommended for integration tests in the perspective of the system as a whole. With respect to the layer, however, any integration testing method is suitable as long as it is beneficial to the development.

Whenever a service user added to the system derives from the component already present in the system, regression tests ensure that the old functionality is preserved. Such testing, however, may be more difficult to perform than in case of service providers. This is due to the fact that service users often provide a graphical interface for the end user, which is rather complex and labour-intensive to test automatically [111] [180].

6.5. Acceptance tests

Unit tests for a service provider ensure that a proper service is implemented according to the requirements. *Acceptance tests* serve the same purpose with respect to the service users. Furthermore, these tests can be used to verify that certain non-functional requirements are implemented properly.

In order to support the development of both projects we presented, *Trac* [51], a free, open-source issue tracking system was used. The system provides a graphical front-end to the source code repository and allows collaborative cooperation on text documents.

The main focus of *Trac*, however, is on the representation of development issues as *tickets*. Each ticket is characterised by a number and a name, followed by a description of functionality and custom type. Additional properties allow assigning a ticket to a particular component or a release version. Furthermore, the issues can be characterised and organised by keywords. An example of such document, taken from the development of our large case study, *BioImageXD2*, is given in Figure 17.

The issue tracking system integrates with SFI development process seamlessly. In particular, tickets of different types corresponded to features, components, layers and customer requirements.

The requirements of *BioImageXD2* and each of the components of the system were discussed with the stakeholders during frequent meetings. The results of such discussions were used to form a detailed description of a ticket that corresponds to a requirement. We aimed at including answers to all the questions and doubts raised by the development team during discussions. Furthermore, we utilised comment system built in *Trac* to support communication during the development cycle. The description of the ticket, together with the comments, was used as a basis for acceptance tests, held at the end of each development cycle.

#64 - Gallery visualisation module	
Reported by: miki	Version: 0.1
Owned by: johan	Keywords: module visualisation gallery slices
Priority: normal	Estimated Number of Hours: 48h
Milestone: gallery visualisation	Billable?: yes
Component: visualisations	Total Hours: 18h

Description

Implement a gallery visualisation module. This visualisation combines both Slices and Gallery modes from the previous version. More details in #65 #66 #67 #68 #69.

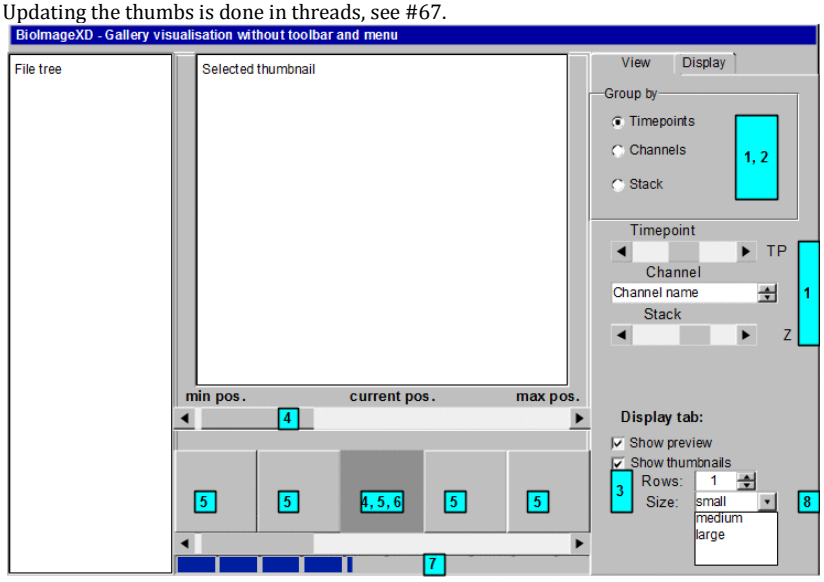
The user interface of this module should more-less resemble what is shown on the picture below. The drawing space of the visualisation contains two parts: preview and thumbs.

The preview shows currently selected thumb. The thumbs are miniatures of all images that belong to current view. When the user clicks on the thumb, the image it represents is shown in the preview. Currently viewed thumbnail is clearly marked. The user has the option to group images by:

- timepoint (with channel and stack specified by the user)
- channel (with timepoint and stack specified by the user)
- stack position (with timepoint and channel specified by the user)

The size of the thumbnails and the availability of preview and thumbs can be specified by the user. See #68 #69. The thumbs are loaded from cache directory; see #65 and #66.

Updating the thumbs is done in threads, see #67.



Notes to the numbers:

1. Changing the option causes the thumbnails and the preview to be updated.
2. Selecting an option from this group disables the corresponding controlling component below.
3. Advanced options, visible to the user only when expert or advanced mode is on.
4. Scrollbar used to control which of the thumbnails is currently shown. The labels indicate minimum, maximum and current position. For timepoints they should be timestamps, for channels – channel names, for stack – numbers. Moving the scrollbar changes the currently selected thumbnail. There can be only one selected thumbnail.
5. Thumbnails are SwitchGlyphButtons. Images are updated in threads, in the background.
6. Currently selected thumbnail that is shown in the preview part.
7. The progress bar that shows how many of the thumbnails have already been displayed.
8. The size of a thumbnail. Changing it causes the buttons to be reloaded. Different thumbnail sizes should use different cached images.

Figure 17. An excerpt from the issue tracking system.

6.6. System testing

System testing, as the name implies, tests a completely integrated system to verify that it meets its requirements [169][181]. It falls into the category of black-box testing [169] and should not require any knowledge about the design, implementation or inner logic of the system. System testing is performed as the final step of the development process, when the individual components and connections between them have been tested.

The system test typically requires a plan that organises test cases and groups them according to the component of the system they refer to. Each and every functional requirement of the system should have at least one component that realises it and a number of test cases ensuring that this realisation is correct. The system test plan should not be a redo of the unit tests, but instead it should be written with a less code-oriented approach. Moreover, the plan should not focus on specific pieces of code logic; instead the functionality of a system as a whole should be reflected in the plan [165].

The aim of system testing is thus not to test the design or the implementation, which are tackled in earlier stages of the development. Rather, the goal is to check the behaviour of the system and its compliance with the requirements. System testing should also be used to verify the expectations of end users, the graphical user interface and security vulnerabilities among others [21].

In case of large and complex software systems not all the functionality of the system can be covered with tests. The difficulty can be additionally increased if the required functionality of the system deals with image processing or displaying, as it was the case with *BioImageXD2*. Certain characteristics of an image could have been checked without showing it, like the number of distinct colours or the dimensions. However, in most cases in order to evaluate whether the software works as required, the processed image needed to be displayed on screen and compared with the expected output.

For the abovementioned reasons *BioImageXD2* was frequently executed during the development, in particular after a significant modification of the image processing code of any module. Due to the principles of SFI, the software stayed executable after each development cycle. The separation between modules and the dedicated system executable further contributed to allowing an incomplete code to be run.

Frequent execution was made an integral part of the coding process, as shown in Figure 18. This approach allowed us not only to ensure that different parts of the system interact properly, but also to improve the graphical user interface.

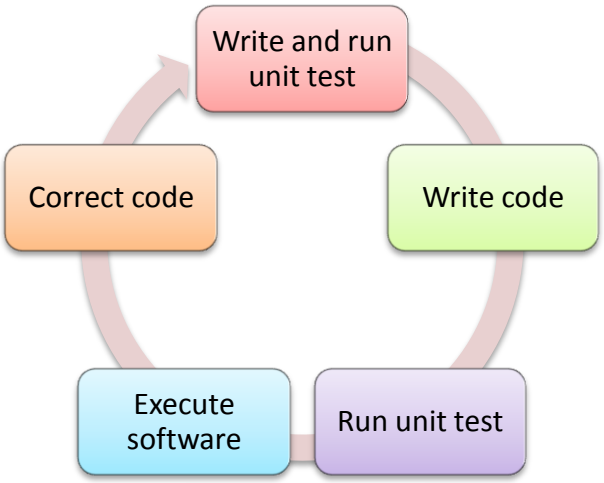


Figure 18. Software execution as part of coding process.

7.Agile Development Process for SFI

Due to its iterative nature, SFI fits well agile development processes, in particular Extreme Programming described previously. In our work we used another agile development philosophy, Scrum [154], instead of XP. Scrum is not a process or a technique for building products; rather, it is a framework within which various processes and techniques can be employed. The role of Scrum is to increase the relative effectiveness of one's development practices so that one can improve upon them, while providing a framework within which complex products can be developed [155].

7.1. Scrum

Certain practices of Scrum follow other agile development methods, like Extreme Programming. Both methods rely on frequent releases and short development cycles; moreover, process improvement is an integral part of both Scrum and XP. In addition Scrum introduces several concepts on its own [159]. The most distinguishing ideas of Scrum are the roles of the stakeholders, the organisation of sprints and a defined set of meetings to be held as the development continues.

Roles

Scrum is a process skeleton that contains sets of practices and predefined roles which fall into one of the two distinct groups – *pigs* and *chickens* [153]. The latter includes all interested in the project, but indifferent to whether the project succeeds or fails, e,g, final users, vendors, etc. Pigs are those committed to deliver the software and taking the blame in case of failure:

- *The Scrum Master*, who maintains the processes (typically a project manager);
- *The Product Owner*, who represents the stakeholders and the business;
- *The Team*, a cross-functional group of people who do the actual analysis, design and implementation.

The sole responsibility of the Scrum Master is to ensure that there are no obstacles for the Team to deliver the product. In other words, Scrum Master is responsible for managing and maintaining the development process and for providing necessary resources for the Team. This role is typically given to the project manager.

The role of the Product Owner is given to the representative of the stakeholders and the customers. It usually is the customer, its designated representative, or an executive of the company that produces the software, i.e. a person that finances the development. As such, a person in this role has a final word in discussions regarding the functionality of the product. Contrary to practices of Extreme Programming, the presence of the Product Owner is not required during the development process. Other means of contact must be assured if the Product Owner is not constantly available.

The Team is responsible for implementing the software according to the requirements of the Product Owner. It should ideally be formed to achieve cross-functionality in the area of software development. Programmers, designers, architects, product-line managers, and testers typically form one or more Teams that are responsible for delivering the product. Moreover, the Teams are self-managing and collectively responsible for the work done. The decision about the number of members in each Team is left for the Teams themselves; however, any team over 7 in size should be split [37] [168].

Sprints

As other agile development methods, Scrum advocates frequent, iterative releases of the code. During each *sprint* the Team creates a potentially shippable product. The overview of the process is given in Figure 19 [182].

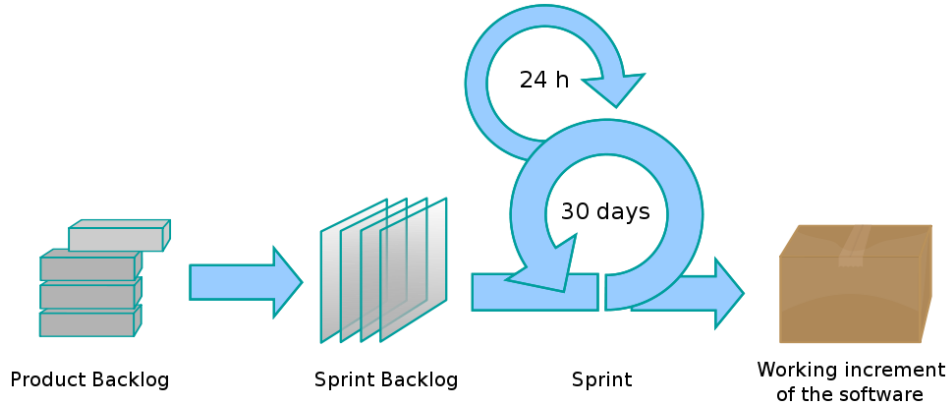


Figure 19. The overview of Scrum sprint.

The set of features to be realised in a sprint is taken from the *product backlog*, which is a prioritised set of high-level requirements. At the beginning of each sprint the Product Owner informs the Team which items need to be completed. The Team then decides how much they can commit to realise during the cycle. After the goals for the sprint are decided, they are moved to the *sprint backlog*. The contents of this backlog remain unchanged for the duration of the sprint.

The sprint has a fixed duration (usually two to four weeks) and must end on time. Any items that are left unimplemented in the sprint backlog are returned to the product backlog at the end of the sprint. After the sprint is completed, the Team is responsible for presenting the developed software [183][153].

During each day of the sprint the Team is responsible for implementing the items from the sprint backlog. Furthermore, the Team decides the details of such implementation and has full control over the source code.

Meetings

Scrum relies on frequent communication, similarly to other agile development methods. A number of meetings is organised throughout the cycle, as shown in Figure 20.

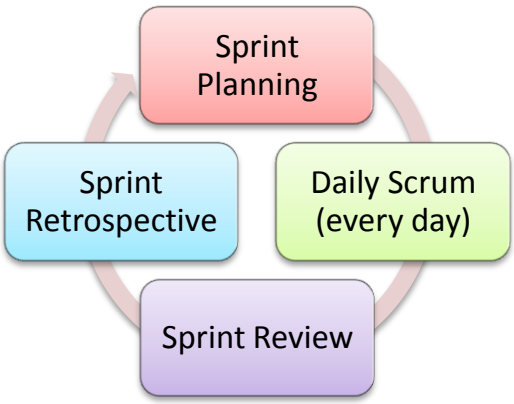


Figure 20. Meetings during a sprint in Scrum.

A *Sprint Planning Meeting* is held at the beginning of each sprint to decide what items from the product backlog should be included in the sprint backlog and how the work should be organised. The Product Owner assigns priorities to the items in the product backlog. Based on that information and the experience the Team decides which items are to be implemented during the sprint. Both the priorities and the selection of items can be changed during the meeting as a result of discussion between the Product Owner and the Team.

A *daily scrum* – a stand-up meeting limited to 15 minutes – takes place every day at the same time and place during the sprint. Only the Team and the Scrum Master are allowed to speak. Each Team member must answer three questions:

1. What have you done since yesterday?
2. What are you planning to do today?
3. Do you have any problems preventing you from accomplishing your goal?

At the end of each sprint two meetings are organised. *Sprint Review Meeting* focuses on the work that has been carried out during the sprint and discusses sprint backlog items that have been completed (*done*) or not. The Team is expected to present an executable version of the system to the stakeholders during this meeting. Following is the *Sprint Retrospective* which is intended to improve the process. Two main questions are asked during this meeting: *What went well in this sprint? What could be improved in the next sprint?* All the above mentioned meetings are time-boxed (eight hours for Sprint Planning, four and three hours for Sprint Review and Sprint Retrospective) [153].

7.2. Adapting Scrum

The analogies between Scrum and Stepwise Feature Introduction presented in Table 6 are straightforward. The sprints are equivalent to iterations and the items included in the backlogs directly correspond to different types of features. The product backlog contains usually requirements, as it is decided in cooperation with the Product Owner.

During each Sprint Planning Meeting a number of requirements is decided to be implemented in the iteration and therefore moved from product backlog to sprint backlog. The Team then decides on how to realise them and proposes a number of components and layers, which are also added to the sprint backlog.

Each day of the sprint the higher-level features are realised by implementing classes. These classes, together with the layers they belong to, form common vocabulary used during the daily scrum for communication between Team members. Thus, the purpose of the sprint is to deliver a number of decided requirements, components or, in rare cases, layers, by implementing a number of classes. We will cover these processes in more details in the following sections.

Scrum	Stepwise Feature Introduction
Product backlog items	Requirements, sometimes components
Sprint backlog items	Requirements, components, layers, sometimes classes
Sprint	Iteration
Product Owner	Customer, stakeholders
The Team	Development team
Chickens	Users, vendors

Table 6. The analogies between Scrum and Stepwise Feature Introduction.

Scrum *per se* does not provide any techniques, being only a framework in which different processes and techniques can be utilised. Integrating SFI with this framework is almost effortless, as it matches common concepts of Scrum directly. As a result one can benefit from agile development philosophy and, at the same time, provide a well-organised and carefully designed architecture for the software under construction. In the following sections we focus more on how SFI can be merged with a Scrum-based development processes.

7.3. Introducing functionality

Scrum is made of repeatable iterations called sprints. The Sprint Planning Meeting is held at the beginning of iteration and allows the Product Owner to prioritise the customer-level features. The meeting is time-boxed to eight hours (or one working day). We propose to divide the meeting into two parts, named after the features that are decided in each of them: *customer-centric* and *architecture-centric*. Both parts of the meeting should be time-boxed to four hours (or half of the working day); however the limits may be adjusted according to the experience level of the Team and the Product Owner. It is also possible to organise a number of customer- and architecture-centric parts one after another, provided that the total time does not exceed one working day, both types of parts are organised equal number of times and for equal amount of total time and the customer-centric part always precedes the architecture-centric one.

The customer-centric part of the Sprint Planning Meeting starts with the Product Owner prioritising the requirements that are left in the product backlog. The priorities are assigned before each sprint, therefore it is possible for one item to be of high importance in one iteration and not important in the other. Moreover, new items can be added by the Product Owner to the backlog at any time; also the existing requirements may be modified between the meetings. However, once a backlog item is decided to be delivered during the sprint, it cannot be modified anymore and its priority remains fixed for the duration of the cycle. The customer-centric part of the Sprint Planning Meeting provides a possibility to make changes to the high-level requirements. It is also an occasion for the Product Owner to introduce new functionality to the product.

The requirements decided during the first part of the meeting are discussed by the Team during the architecture-centric part. The customer is not directly involved in the process – the role is limited to providing feedback during the analysis of the requirements. As said previously, components of the system are affected by the requirements; therefore it is software architect and product-line manager (both being Team members) who are in charge of the component creation process. The components are also assigned priorities, although this depends heavily on what prioritisation has been done by the Product Owner. No component may be of higher priority than the requirement it derives from.

The purpose of this part of the meeting is to decide a number of components to be delivered during the sprint. In other words, the Team must decide how the requirements are going to be implemented and what design is the most beneficial for the product at the current stage of the development. The Team may also decide on refactoring already existing components, if the analysis of the requirements shows that it is more beneficial to the project than delivering new code. Thus, the architecture-centric part of the meeting serves as an opportunity for the Team to add or modify the architecture. The components may additionally be modified during the sprint, as long as they do not affect the functionality required by the customer. The main advantage of deciding the architecture during the Sprint Planning Meeting is that the Product Owner is available and may provide additional feedback instantly, which is not always the case during the sprint.

At the end of the Sprint Planning Meeting the sprint backlog is created. The requirements decided during the first part of the meeting are removed from the product backlog and moved to the sprint backlog. Both backlogs are also updated with the components created during the second part of the meeting. During the sprint only the components may be altered or added, depending on the decision of the Team. The requirements remain fixed, unless both the Product Owner and the Team decide otherwise.

The characteristics of the design (i.e. the layers) ideally should be created at the Sprint Planning Meeting, as soon as the components are decided and the ideas about realising them emerge. However, it is also possible that layers are introduced as the sprint continues in order to simplify the design or to improve its quality. During the sprint there is no specific moment at which a layer should be created or modified. Regardless of such moment the Team must be notified about it, i.e. the knowledge on the design is explicit to the Team. The designers, the programmers and the testers are involved in this process, but the Product Owner is excluded. The layers must be placed in the sprint backlog as soon as they are created and they can be modified or removed as the sprint continues.

The classes implement the requirements and are created according to the current needs (*on the fly*) during the sprint, as soon as one or more Team members accept a requirement or a component to work on. They are introduced solely by the programming members of the Team, with the help of the testers.

The classes should be used for communication during the daily scrum meeting, together with the layer they are part of. It is not required for a class to be placed in the sprint backlog; however it is recommended to keep track of them and their respective layers for future reference, e.g. in case of software failures or code quality issues.

The introduction of functionality with Scrum framework and Stepwise Feature Introduction is summarised in Table 7.

Element (Level)	Introduced by	Introduced at	Item in backlog
Requirement (Customer)	Product Owner	Sprint Planning Meeting (customer-centric part)	Product, sprint
Component (Architecture)	The Team (architect, product line manager)	Sprint Planning Meeting (architecture-centric part)	Product, sprint
Layer (Design)	The Team (architect, designer, programmer)	Sprint Planning Meeting (architecture-centric part), during the sprint	Sprint
Class (Code)	The Team (programmer)	During the sprint	Sprint (optional)

Table 7. Introducing functionality with Scrum.

7.4. Evaluating the implementation

The requirements of a software system usually change during the development. The main benefit of Stepwise Feature Introduction is the constant improvement of the system based on the evaluation of every step. Such evaluation does not anticipate change; instead it helps to react to it in shortest possible time, thus minimising its negative effect and costs. An adaptive software process, such as Scrum, provides mechanisms for performing the evaluation of requirements, components, layers and classes.

Scrum requires a precise definition of the term *done* that is applied to completed backlog items. Such definition should list all the conditions that the item must satisfy before it is considered finished. Only those items that are done may be presented to the customer at the Sprint Review Meeting. Stepwise Feature Introduction, on the other hand, requires that the classes must be correct. The relation between these two terms is straightforward – for a class being correct is a necessary condition to be

done, but not a sufficient one (e.g. the code may lack the documentation, despite being fully implemented and tested).

In this section we propose when to evaluate requirements, components, layers and classes in Scrum-based development process. The main goal of the evaluation is to ensure that the code is not only properly implemented, but also done according to the requirements and quality standards. Each evaluation phase should also include some additional project-specific activities set by the project manager, software architect or the quality assurance.

The review of a class (code that implements some desired functionality) is performed as soon as its implementation is finished, i.e. during the sprint. The evaluation includes executing unit and regressions tests for the feature; additionally a careful code review may be done. In case the tests fail or the code is of poor quality, the class must be re-implemented or restructured before the evaluation is performed again. The results of the evaluation should be communicated back to the responsible programmers or to the Team. When the class is done, it should be used during the daily scrum meeting to answer the question "*What have you done since yesterday?*".

Each layer of the design is evaluated by the Team at two different occasions. The first review must be performed when all its classes are done, which happens during the sprint. A number of integration and regression tests must be performed in order to ensure that the classes function properly together and form a layer. Moreover, an analysis of the dependencies between different classes must be done to secure high quality of the design.

The second evaluation of a layer takes part during the Sprint Review Meeting, which we propose to divide into three parts: design-, architecture- and customer-centric. The parts should be time-boxed to about 10%, 20% and 70% of time allocated for the whole meeting (typically 25, 45 and 170 minutes). The participation of the Product Owner in the design-centric part of the meeting is not necessary; it is however recommended in the second part and required in the last.

The *design-centric* part of the meeting focuses on the layers that have been delivered during the sprint. Each of them is reviewed in the context of how it interacts with other layers of the same component. The design mistakes corrected during the sprint should be also briefly discussed to avoid repeating them in the future sprints. The evaluation is performed

solely by the Team and includes executing or reviewing the integration and regression tests together with an analysis of code quality metrics, particularly those regarding the layer dependencies.

The second part of the Sprint Review Meeting is *architecture-centric*. All components scheduled for the sprint are evaluated, whether they have been completed or not. The review of the delivered components is done by the architect and the product-line manager with the help of higher-level integration tests and system tests. The components that were not completed during the sprint must also be analysed. The reasons for not delivering them must be clearly identified by the Team, so that in the following sprints the errors are not repeated. This evaluation process is internal to the Team; however the Product Owner should be informed about the results, especially when evaluating the components connected with high-priority requirements.

The longest part of the Sprint Review Meeting, *customer-centric*, is carried out as last. The Team is responsible for presenting a working version of the system to the Product Owner. Moreover, the requirements planned for the sprint are evaluated, regardless of their completion. The evaluation is performed only by the Product Owner. The role of the Team is limited to reporting their work and providing motivation for why certain requirements have not been implemented and what difficulties arose during the implementation of the completed ones.

The results of such evaluation not only affect the priorities of other requirements, but also may cause them to be modified or even removed from the final functionality of the system. It is also possible that the evaluation will change the direction in which the system is evolving by causing new requirements to be added to the product backlog. Due to the layered structure of the constructed software it is possible to remove one or more recently added layers in order to continue development in another direction. Moreover, it does not require any additional work to restore the software to an executable state, as each layer (together with its lower layers) forms an independently executable system.

The iterative nature of both Stepwise Feature Introduction and Scrum allows evaluating requirements, components, layers and classes as soon as they are added or modified. Based on the results of such assessments the decisions regarding the implementation and the design of particular requirements are made. It is possible for a class or layer to

extend, combine, or even replace one or more previously existing ones, depending on which solution is the best alternative at given moment.

It should also be noted that the components and layers must encapsulate the best possible solution for the requirements selected for the particular sprint. Moreover, the design should allow possible future changes, as anticipated by the Team, but without relying on the requirements that were not picked for the current sprint.

The constant evaluation of the system, which we summarise in Table 8, may help in achieving the compromise between agile-specific *design to suit current needs only* and the *big design up front* of the traditional development processes [163] [183].

Element (Level)	Evaluated by	Evaluated at	Remarks
Class (Code)	The Team (programmers, testers)	During the sprint	Part of daily scrum communication
Layer (Design)	The Team (architect, designer, testers)	During the sprint and at Sprint Review Meeting (design-centric part)	Part of daily scrum communication
Component (Architecture)	The Team (architect, product line manager)	Sprint Review Meeting (architecture-centric part)	Product Owner should be informed about results
Requirement (Customer)	Product Owner	Sprint Review Meeting (customer-centric part)	Not done features are also evaluated

Table 8. Evaluating functionality with Scrum.

7.5. Process summary

The V-Model, shown in Figure 21, is a system development model designed to simplify the understanding of the complexity associated with developing systems [59]. It is a graphical representation of the development lifecycle and summarises the main steps to be taken. The left side of the diagram *defines* the project, while the corresponding actions that *verify* it are shown on the right.

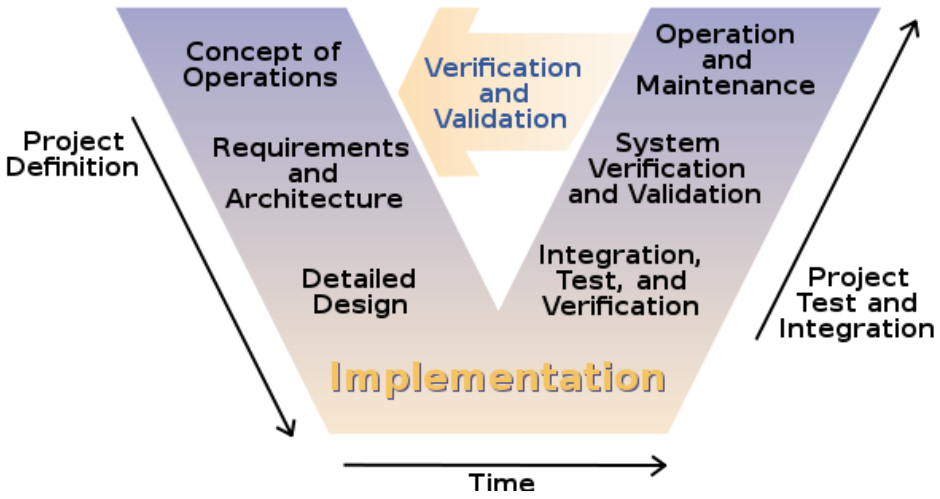


Figure 21. The V-Model of a typical system development.

The SFI-Scrum-based development process – more precisely, each sprint – can also be presented as such structure, shown in Figure 22. The left side of the diagram shows the *introduction* of functionality, while the *evaluation* steps are presented on the right. The level of details changes from more general (requirements) to more detailed (code) as the functionality is introduced and implemented and follows the reverse direction during their evaluation.

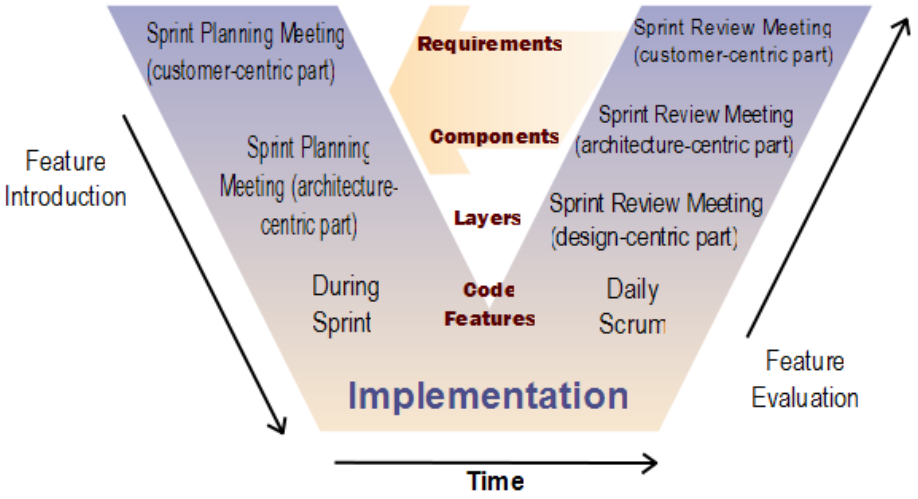


Figure 22. The V-Model of a sprint in Scrum-and-SFI-based development process.

Part III:
Case study – ReThink

8. Design

In order to illustrate the basic concepts of SFI, we will now present a non-trivial case study, an interactive board game. The software was developed as part of a project course held at Åbo Akademi University. The development team was formed from students of computer science and software engineering.

8.1. Rules and history

Think! Is a turn-based board game for two players. It is played on a square, 6-by-6 board, shown in Figure 23. The board is initially empty. Players make their moves in turns by sliding their counters onto the board from bottom or right, whichever side they choose. The contents (i.e. the counters and the empty fields) of a row or column affected by the move are shifted accordingly, with the top-most or left-most counters falling of the board and being removed from play. Each player must make a move during his turn. The goal of the game is to construct a line of four own counters, horizontally, vertically or diagonally. Such line may also be created as a side-effect by the opponent [172].

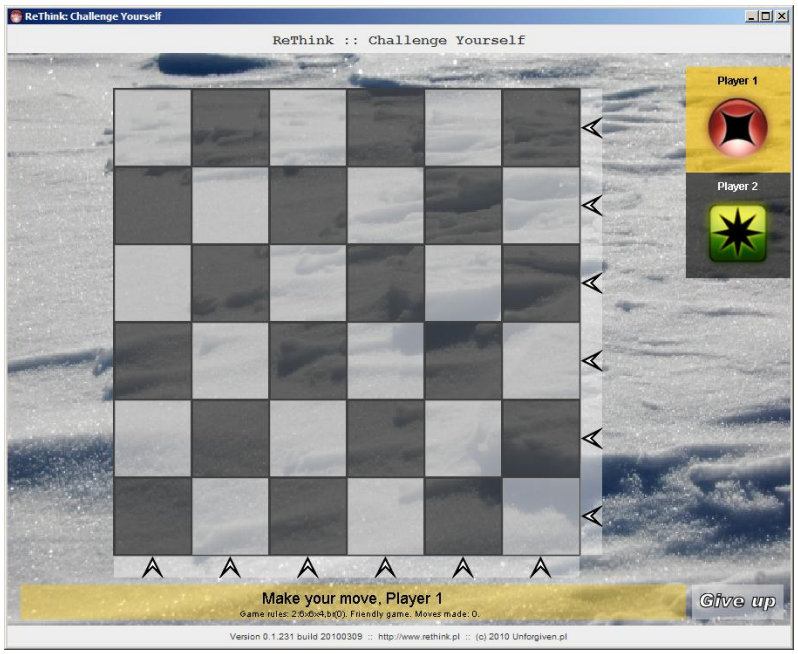


Figure 23. An empty board in the PC version of *ReThink*.

Figure 24 (left-to-right and top-to-bottom) shows an example of a game between two players using black and orange counters, respectively. A colour-matching indicator is used to show a move a player is about to make.

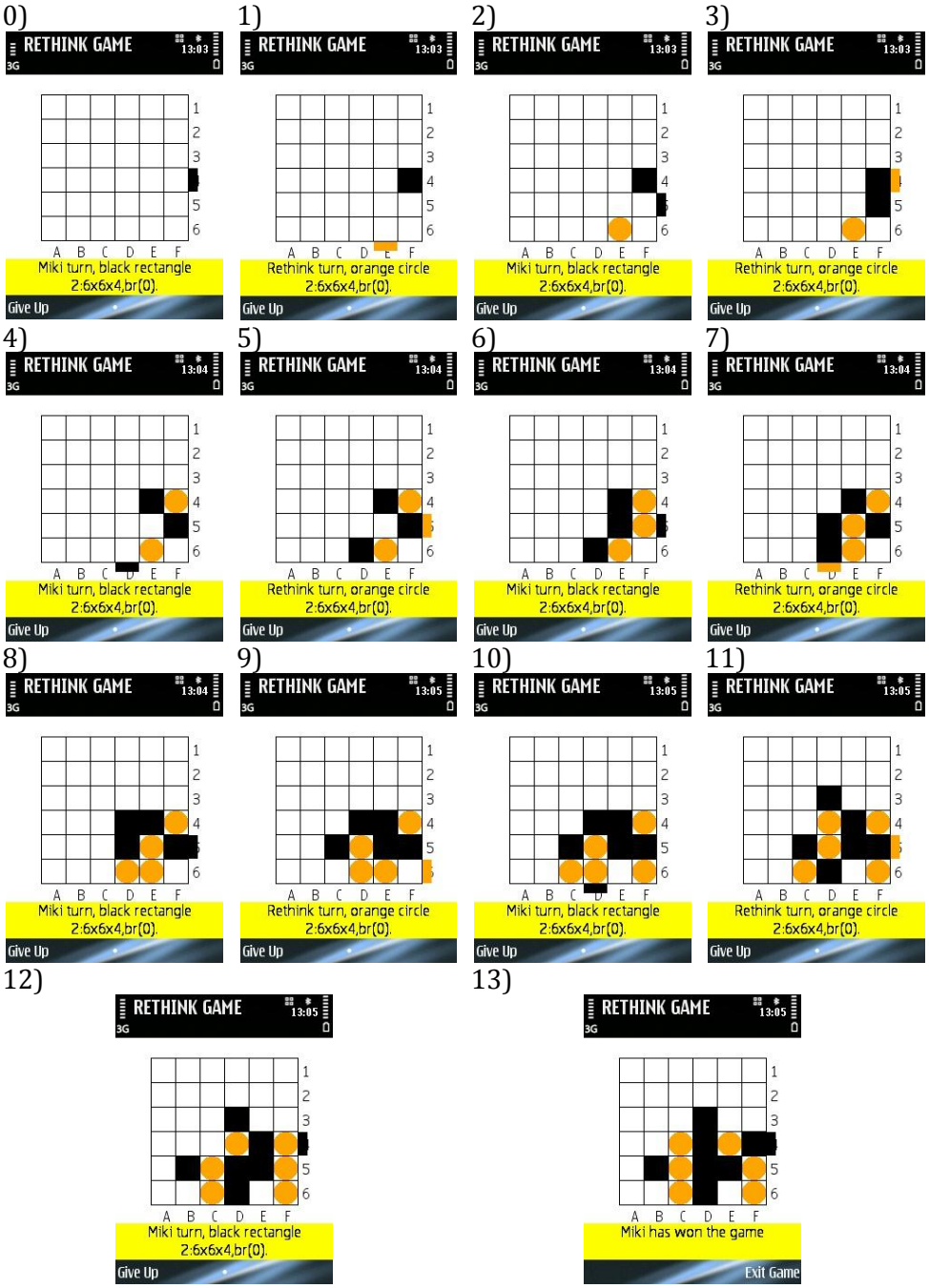


Figure 24. A complete game play of *ReThink*, as seen in version for mobile phones.

With a total of 13 moves Figure 24 depicts what is a typical game. The first player (playing black counters) attempts to win by constructing a diagonal line F3-E4-D5-C6, a strategy discovered by the opponent after move number 7. A series of moves that foresee and void the strategy of the counterpart leads to a situation (move 12), which the player using orange counters did not anticipate. This left the opponent with an easy win.

8.2. Requirements

The game of *Think!* has been originally published in 1985 by Ariolasoft and released for ZX Spectrum microcomputers. Designed by Chris Bishop, Chris Palmer and Beth Wooding, the game met with positive reception by the critics, being described as "*demanding, intriguing game*" [25] and "*easy to learn, [but] a devil to master*" [71]. It was also mentioned in the Official All-Time Top 100 Spectrum Games at place 29, the top for board and puzzle games [36].

We decided to develop a modern remake of this *cult classic* while keeping the basic rules unchanged, hence the name *ReThink*. It has also been agreed that the game will be published by an independent software company and distributed by a manufacturer of tablet computers. Together with the representatives of the companies we decided that the software should:

a) allow its users to play the game according to the classic rules, but provide mechanisms for extending them;
b) be deployed on a number of platforms (dedicated board-game hardware, mobile phones, desktop computers and web browsers) and support touch-screen devices wherever appropriate – implying code reuse and modularisation;
c) offer an offline multiplayer mode to enable games between players sharing the same device;
d) allow cross-platform online games over the internet to provide players with a unique experience and the possibility of playing the game at any moment – indicating existence of a game server;
e) have an online ranking system, similar to the ones used in chess or go, and updated after every game to help in building web community of players.

From the customer point of view these characteristics of the system are its features – well-defined parts of the desired functionality. Therefore, a feature as understood by a customer is a real, identifiable *requirement* of

the system. A stepwise introduction of requirements requires a detailed evaluation and elaboration on the interaction between them, combined with a careful planning and design of the software architecture.

The requirements mentioned above are tightly connected and dependent on each other. For example, an online ranking system can be introduced to the system after the online games are made possible, part of which can be developed in parallel with the offline multiplayer mode.

In order to minimise the development effort and reduce the number of potential defects the code should be shared between different deployment platforms – ideally, only the graphical user interface should be platform-specific. Such constraint implies that all the other features are introduced to the system earlier, before the different deployment platforms are implemented. Moreover, the basic mechanics of the game should be added to the system at the very beginning of the development. The ranking system, on the other hand, requires that the game server is working and that at least one deployment platform has a playable version of the game. The above considerations affect the final order of introducing the requirements, as shown in Figure 25.

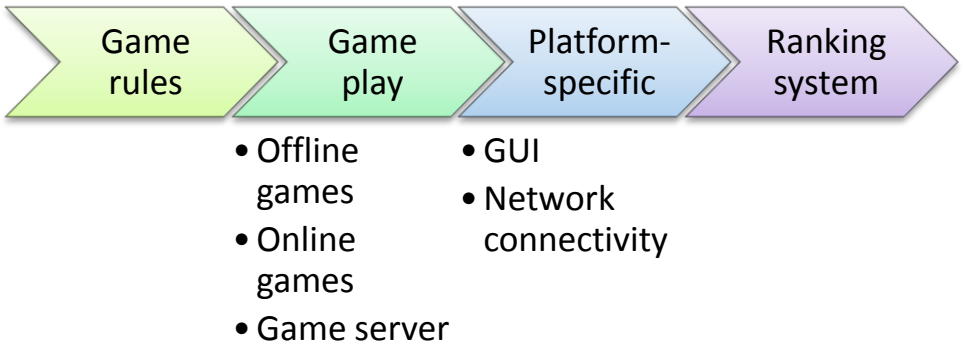


Figure 25. Introducing requirements to *ReThink*.

In addition to the above requirements, we decided to develop a simple text-based GUI to use for demonstration purposes. Its code was expected to serve as a basis for development of GUIs on other platforms. Furthermore, its early availability helped to identify certain flaws in the mechanics of the game and lead to removing a number of requirements initially thought of as useful.

8.3. Components

The requirements specified in the previous section are well-defined, but abstract and do not contain information on how they should be designed and implemented. This is decided at the architectural level, where requirements are refined and analysed by a system architect and become part of the design.

The requirements show what to achieve with the software system, the architecture describes how to do it. It determines the basic blocks with which the complete system is built; therefore we refer to them as *components*.

An analysis of the requirements allows us to identify the following main concepts of the system:

- *Room*: a container for games. Different players can submit or cancel their games in the room.
- *Game*: a game session between players. Each game has clearly defined rules and a state.
- *Rules*: define set of rules for a game. The rules are responsible for creating the game's initial state, as well as deciding a winner or next player, based on any of the game's states. Moreover, the rules are providing a number of states that are possible to reach from any given state.
- *State*: a snapshot of a game situation at any moment. Before the game starts, there is no state. Players can change state of the game in turns, according to the rules of the game.
- *Player*: someone participating in a game. A player can join the game or leave it. A player can also host a game, which means it is possible for that player to control the rules of the game and decide which of the other players will play the game.
- *Board*: a game board, with players' counters on it. It is an integral part of a game's state.

All the above concepts can be represented in the software as interfaces. The relation between them is shown in Figure 26 (declarations of methods and attributes are omitted for clarity).

It can be noticed that the interface `Game` is the central concept that affects or is affected by the others. We can also notice that the interface `Game` is the service provider for the interface `Room` and at the same time is using services of `Rules`, `State` and `Player`. This relation implies the order

in which these concepts can be implemented and introduced to the system. Further analysis reveals that the classes related through composition (`Game`, `Rules` and `State` with `Board`) are tightly connected with each other and should be introduced to the system in parallel.

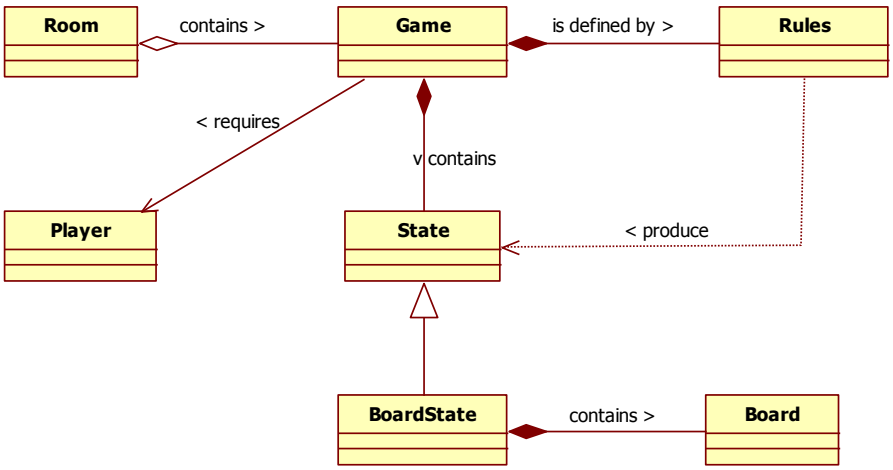


Figure 26. Relations between main concepts of *ReThink*.

The above concepts thus form four distinct architecture components, as shown in Figure 27:

a) Players (Player);
b) Game Playing (Game)
c) Game Mechanics (Rules, State and Board);
d) Game Rooms (Room).

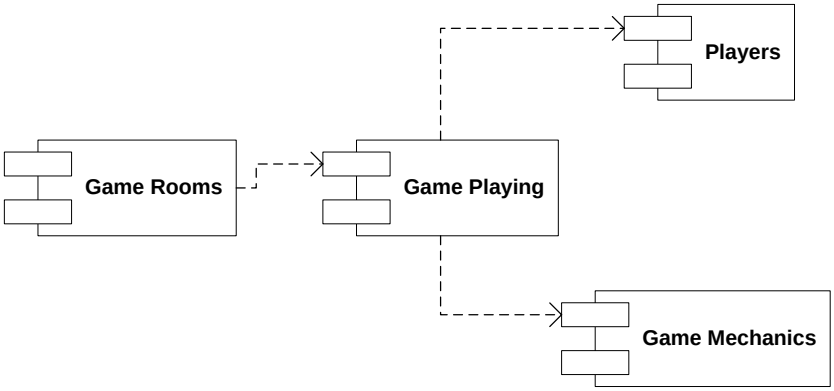


Figure 27. Relation between components in *ReThink*.

The introduction of these components – in case of *ReThink* – has been done by gradually adding the interfaces and specifying their methods. Together they form an overall architecture on which the software is built. Each of the components helps in realising one or more requirements. The distinction between Game Playing and Game Mechanics is a design decision: to maximise the possible future value of the project we decided to allow playing games based on mechanics different than the ones of *ReThink*.

The component Players holds one interface that defines the properties of a game player, which allows storing and exchanging player information in multiplayer modes regardless of the deployment platform. Moreover, it is required to actually play the game. Both Game Mechanics and Game Playing contain interfaces that allow playing the game according to the rules, encapsulated in a separate interface. Finally, Game Rooms provide an abstraction common to both online and offline multiplayer games.

As it can be noticed the availability of software on different deployment platforms is missing from the design considerations. The software built with SFI must be executable at each stage of the development. For *ReThink* we decided to use dedicated executables for each deployment platform. Such executable is responsible for constructing and managing the graphical user interface proper for the platform. Furthermore, we decided to design and develop network connectivity separately from the game itself. This approach allowed us to isolate the concerns not directly related to the game from the design and development of the system.

8.4. Classes

The components described previously are defined by interfaces; therefore they are independent of the deployment platform – the precise behaviour is left for the subclasses to implement. Let us focus now on the implementation of *ReThink*. The excerpt from the class hierarchy related to Game Mechanics is presented in Figure 28.

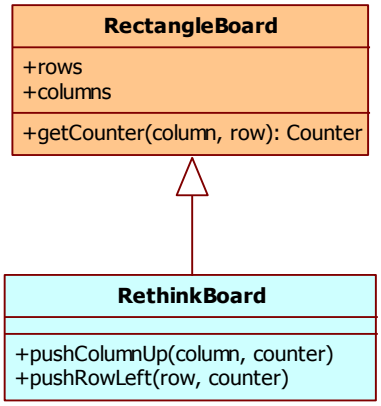


Figure 28. Class hierarchy for RectangleBoard and RethinkBoard.

The class `RectangleBoard` descends directly from the interface `Board` described previously. This class represents a rectangular board with given number of rows and columns. Each field may or may not contain a counter. The class provides a method for checking what counter is located at given board coordinates (row and column). We do not allow the class to change the arrangement of counters on the board – the initial setup must be decided at construction time.

The subclass `RethinkBoard` introduces new behaviour to the board. With this object it is possible to change the arrangement of the counters by sliding a counter onto the board from one of the board's sides. This class extends all the features of `RectangleBoard` and can be used in the cases where the original functionality is expected. A unit test proper for the `RectangleBoard` would yield identical results when used for the class `RethinkBoard`.

The ability to slide counters onto the board has been encapsulated in a class added to the existing hierarchy. By a feature we understand a class that provides such increment in functionality. In Figure 28 we have shown two such features – the class `RethinkBoard` is a feature added to its superclass, which in turn is an initial feature in this class hierarchy. A class does not have to inherit from a previously existing one in order to be considered a feature.

8.5. Programming language

ReThink is written in Java [167], as this programming language offers the much-needed ability to deploy software to a number of platforms using the

same source code. Furthermore, the development team was familiar with Java and felt comfortable programming in it.

The static typing system used in Java requires that the type of an object is explicitly stated in the source code. As the methods are declared in the interfaces, the precise subclasses are not known. Therefore, the service users that implement these interfaces must refer directly to the type of the provider they utilise. Figure 29 shows an example of extending existing features in *ReThink*. We use inheritance, which is one of the means of achieving subtype polymorphism, to create a hierarchy of service providers (`Games`) and users (`Rooms`). The static typing results in an explicit check to ensure that the service user receives the expected service from the provider. An example can be found in the class `ServerRoom`, as presented in Listing 1 in Appendix 3.

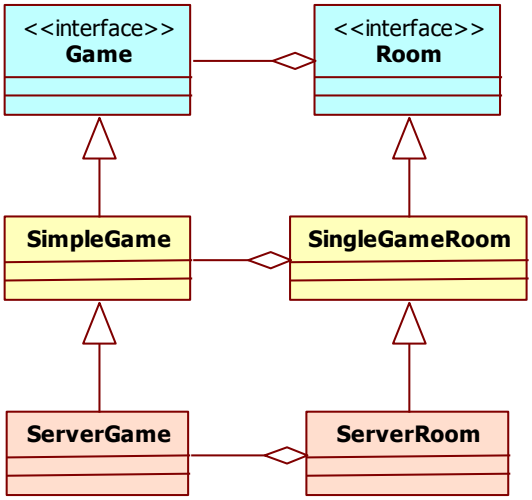


Figure 29. The hierarchy of Rooms and Games in *ReThink*.

9. Layers

Gradual introduction of functionality to the system in construction naturally leads to a layered design. This is an essential characteristic of the paradigm and can be clearly seen in the design of *ReThink*. It must be underlined at this point, that all the requirements of the final system were known in advance. Thus, the development of *ReThink* differed in this manner from the construction of other software systems.

9.1. Introducing functionality

The development started by defining the components of the whole system, followed by division of the overall functionality into small, manageable steps. The first increment delivered a rectangular board, which was later extended with functionality specific to *ReThink*. However, it is only one step in the process of enabling users to play the game with the software. The classes that represent other concepts (e.g. `Rules`, `State`) must also be extended to become aware of the new functionality, as shown in Figure 30. Such extension forms a layer [8] in the class hierarchy. The figure lists four layers, two of which are abstract (red and yellow) and two are concrete (orange and blue). All the classes belong to the same component, i.e. Game Mechanics, with an exception of the interface `Game`, which belongs to component Game Playing.

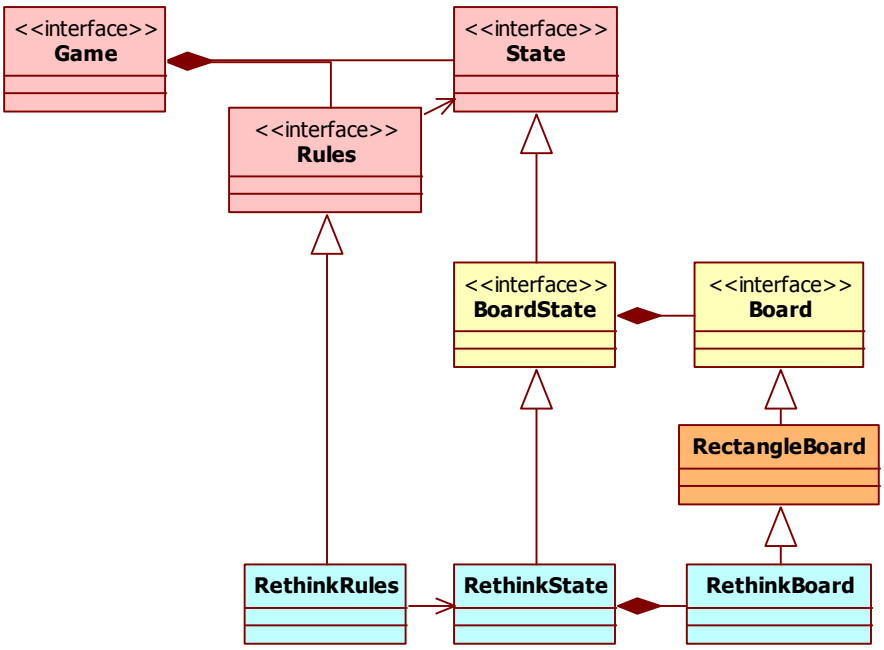


Figure 30. Introducing the rules of *ReThink* to the design.

It can be noticed that on an abstract level interface `Board` is a service provider for `BoardState` and `State` is a service provider for `Rules`. The interface `BoardState`, which extends `State`, is both service provider and user: it utilises the functionality of `Board` and – because of its superclass – provides a service for `Rules`.

In order to effectively use a code feature encapsulated by the class `RethinkBoard` we need to extend the functionality of `State` and `Rules`, hence classes `RethinkState` and `RethinkRules` are created. Each of them is a feature in itself, because it provides new functionality to the system by extending already existing classes. A collection of classes created to utilise a single feature forms a layer in the class hierarchy.

Alternatively, we can say that a layer consists of those classes that directly or indirectly benefit from the same service provider, but are not extensions of it. This definition is not affected by the order in which the classes are introduced to the system. We could implement the classes `RethinkState` and `RethinkRules` before adding `RethinkBoard` to the system, but the resulting layer would be the same due to the latter class being the service provider for the former two.

9.2. Component layering

The requirements of a system are realised by a number of architectural components. These components are built of classes that contain the code, arranged in a layered manner. As a result, the system has an organised and clearly recognisable structure. The overview of the structure of *ReThink* code is shown in Figure 31.

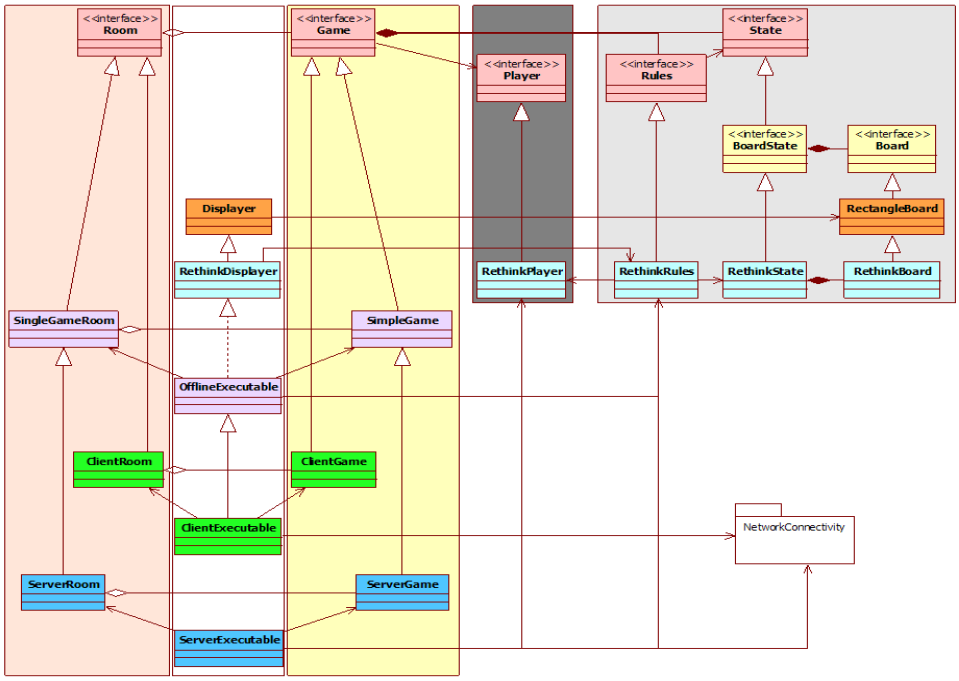


Figure 31. The design of *ReThink*.

The background colour of the classes indicates a layer which they belong to. The coloured vertical bars are used to identify components of the system; from left to right: Game Rooms, Game Playing, Players and Game Mechanics. The white vertical bar is used to identify the executable classes, which may be seen as a separate component of the system.

The two top-most layers (horizontal, red and yellow) are abstract, as they contain only interfaces and declare the functionality of the system. The subsequent layer (orange) introduces only one feature to the Game Mechanics component – a rectangle board. A stand-alone executable class is provided in this layer as well in order to be able to execute the system here.

The third layer (light blue) adds game mechanics specific to *ReThink*. The executable from the previous layer has been extended to support new functionality. The development of the components Game Mechanics and Players, for the purpose of *ReThink*, has finished at this point.

In the fourth layer (light purple) the components Game Playing and Game Rooms receive their initial functionality. More precisely, we use the functionality provided in the previous layer to enable offline games.

Two final layers (green and blue) were developed in parallel and, in fact, form two separate systems. However, both layers together realise one requirement – online games – by relying on external network connectivity package. The green layer provides this functionality from the client perspective, while the blue implements the server. Four more parallel layers that extend the customer (green) layer were added to the system, thus forming four separate subsystems. Each of them was focused on a separate deployment platform (i.e. mobile phone, touch-screen tablet, personal computer and web browser) and extended only the executable to include the specific details, like the graphical user interface and the network connectivity. Since no other classes important to the design were added, these layers are not shown in Figure 31.

An important characteristic of the design is that the layering varies from one component to another. The system, as shown in Figure 31, has seven layers, five of which are executable. The component Game Mechanics contains two concrete layers; Players have only one layer, which is shared with Game Mechanics. Both Game Rooms and Game Playing contain three shared layers, separated from the other components. However, these layers rely on the ones added to the system previously. Moreover, the top-most

abstract layer is common to all components of the system, as it declares their functionality, responsibilities and dependencies.

The system shares properties of various architectural styles. The client-server approach [115] is clearly seen on a higher level – each deployment platform has its own client application. The information with other clients is exchanged through the central server. The implementation of the clients and the server benefits from implicit method invocation, caused by event broadcasting [62]. In turn, we could use asynchronous message processing and allowed a high number of clients to be connected to the server at the same time.

10. Correctness Conditions

The paradigm of SFI incorporates *correctness* as an essential concern in the development. As presented earlier, there are four conditions that must be satisfied before a feature is considered correct: Internal Consistency, Respect, Preserving Old Features and Satisfying Requirements. Furthermore, the paradigm supports and encourages diagrammatic reasoning, which we describe in more details in this section.

10.1. Internal Consistency

Internal Consistency is ensured by identifying and proving class invariants, defined as properties that must be satisfied before and after executing any of the operations the feature provides. Additionally, for each operation additional conditions must be checked. Pre-conditions define under which circumstances an operation can be executed, whereas post-conditions state the properties of the system after the execution has taken place. Proving that the implementation of an operation satisfies all the invariants and the post-conditions when the invariants and the pre-conditions are true initially is required for a feature to be internally consistent.

An example of diagrammatic reasoning for a part of *ReThink* class structure is shown in Figure 32. We start our reasoning from the service provider in the first layer that contains implementation, i.e. the class `SimpleGame`. The conditions for its internal consistency are a result of the system requirements, the component the class belongs to and the role it plays. The class `SimpleGame` belongs to the component Game Playing, thus its primary purpose is to hold a game session between players. Before the

playing starts, however, the player that hosts the game is free to modify the rules and invite other players. Once the game is under way no more changes are allowed and the object is expected to hold the current state of the game.

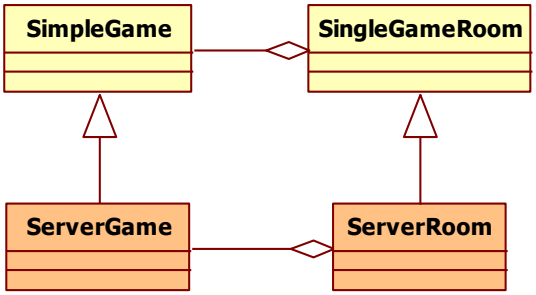


Figure 32. Excerpt from the class hierarchy in *ReThink*.

Similar considerations must be taken into account when reasoning about correctness of the class `SingleGameRoom`. The name of this class implies its most important characteristic: only one game can be stored. In other words, once a game has been submitted to it, no other submissions can be made before that game is finished or removed.

We can ensure that the conditions hold for those classes by testing them, as indicated in Figure 33. However, stating that both `SimpleGame` and `SingleGameRoom` are internally consistent does not automatically mean that the relationship between these classes is correct.

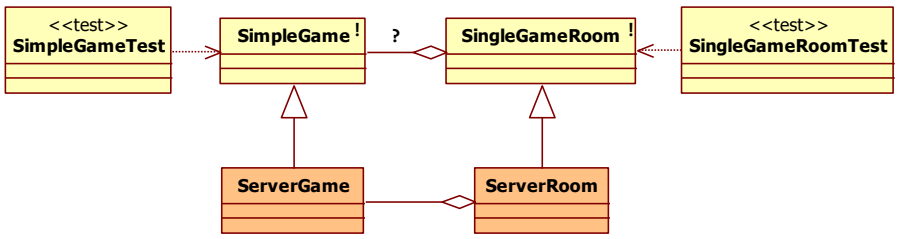


Figure 33. Internal consistency in *ReThink*.

10.2. Respect

The pre- and post-conditions mentioned earlier are also of use when proving that one feature respects the constraints of another feature. Whenever a service user invokes an operation of a service provider, the pre-conditions of this operation must be satisfied. Furthermore, the service provider must establish post-conditions of the called method.

The class `SingleGameRoom` belongs to the component Room. It is a service user and relies on the services provided by the class

`SimpleGame`. Therefore, it is essential to ensure that the constraints of the provider are not violated. Among other implementation-specific details this can be easily covered with tests. Once the tests pass we are allowed to mark the relationship between the classes as correct, as shown in Figure 34.

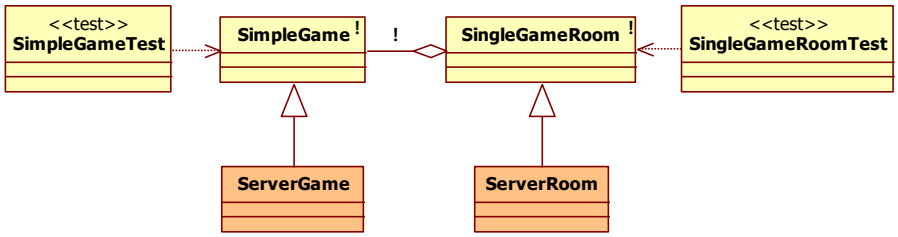


Figure 34. Correctness concerns for aggregation between two classes in *ReThink*.

10.3. Preserving Old Features

The preservation of old features is achieved in a manner similar to ensuring that the Respect condition holds. It must be proven that the added or extended features do not violate the constraints set by the features already existing in the system. Most importantly, the post-conditions of existing feature operations must be preserved and its pre-conditions may not be strengthened.

Two new classes form the next layer of the system, `ServerGame` and `ServerRoom`. The previously described correctness conditions – Internal Consistency and Respect – are straightforwardly handled with tests. The resulting situation is shown in Figure 35.

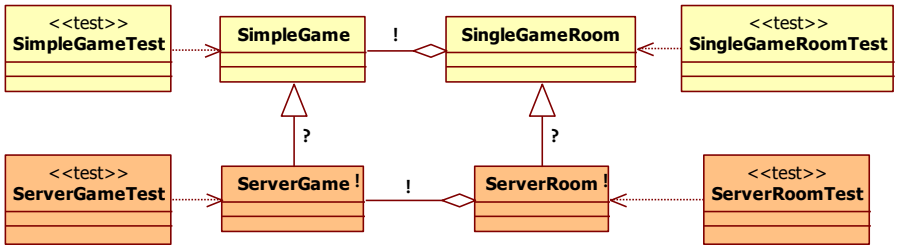


Figure 35. Internal Consistency and Respect for a new layer in *ReThink*.

At this point nothing can be said about the correctness of the inheritance relations between the classes in the previous layer and the newly added ones. To state so we need to ensure that the properties of the superclasses are preserved. We can achieve this with regression testing, as explained

in the earlier part of the thesis. Passing the tests allows us to mark the inheritance as correct, as indicated in Figure 36.

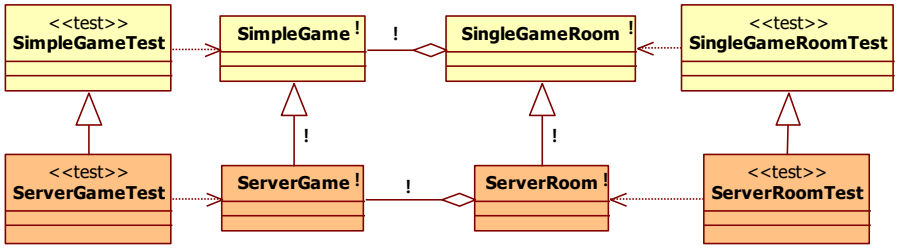


Figure 36. Preserving Old Features in *ReThink*.

10.4. Satisfying Requirements

Software development with Stepwise Feature Introduction is an iterative process. Each of the iterations delivers a well-defined functionality that is presented to the customer to gather feedback. The customer evaluates and accepts each addition to the functionality of the system under construction. This process is intended to guarantee that the customer requirements are satisfied.

The required functionality can also be ensured through testing, provided that the requirements are precise enough. The tests ensure not only that the requirements are satisfied, but also ensure that the constraints of the tested class are respected, as mentioned earlier. Once the tests represent the requirements, are successfully executed and contain no errors, we can annotate them with an exclamation mark, as shown in Figure 37. This indicates that the requirements of the tested classes are satisfied.

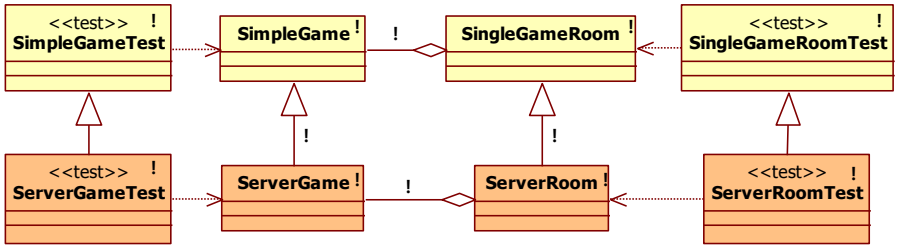


Figure 37. Satisfying Requirements in *ReThink*.

10.5. Correctness conditions for interfaces

Our example started with concrete classes. However, they do not form the top-most layer of *ReThink* – the interfaces specifying the behaviour of the components do. The corresponding diagram is shown in Figure 38.

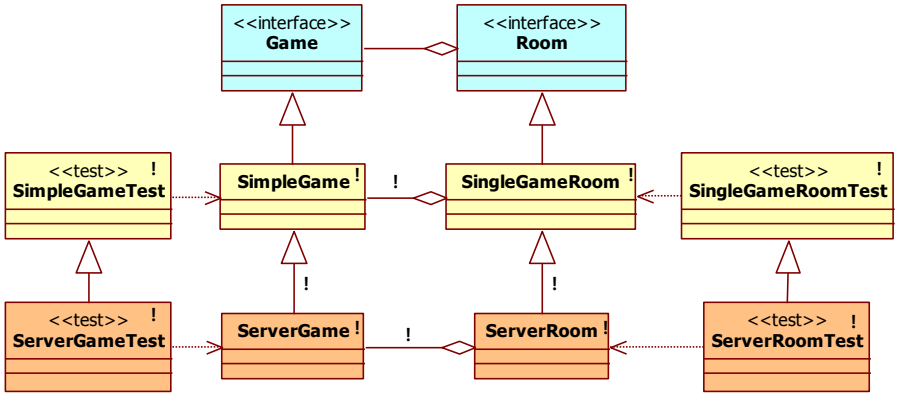


Figure 38. Component interfaces in *ReThink*.

The reasoning about the correctness of the interfaces differs from the reasoning for classes described earlier. Since the interfaces do not contain any implementation, we consider their class invariant to be true at all times.

The set of local variables of an interface is empty. This allows interfaces to be easily extended, as new variables can always be added in the subsequent layers without compromising correctness.

Each method declared in an interface does not carry any implementation (or, in some cases, carries an empty statement that does nothing). The precondition of each declared method is always false. These two properties allow stating that the interface is internally consistent, as the precondition and the invariant establish the invariant once the method is executed.

Since the method precondition is false, it is impossible for any other entity to establish it. Any call to a method contained in the interface is thus not allowed. Instead, it is required for other classes to execute a method of a class that implements the interface. The constraints of the interface are thus respected by other entities that depend on it.

Finally, additional non-formal constraints can be set for any interface. Such requirements naturally affect the classes that implement

such interface. In *ReThink* the correctness conditions were strengthened for those interfaces that aggregated others. More precisely, we required the container interface to declare meaningful methods to manipulate and access the contained objects. In our example these methods are declared by the interface `Room`. Since all other correctness conditions are also satisfied, we are allowed to mark the interfaces and all associated relations with other entities as correct. The final situation is presented in Figure 39.

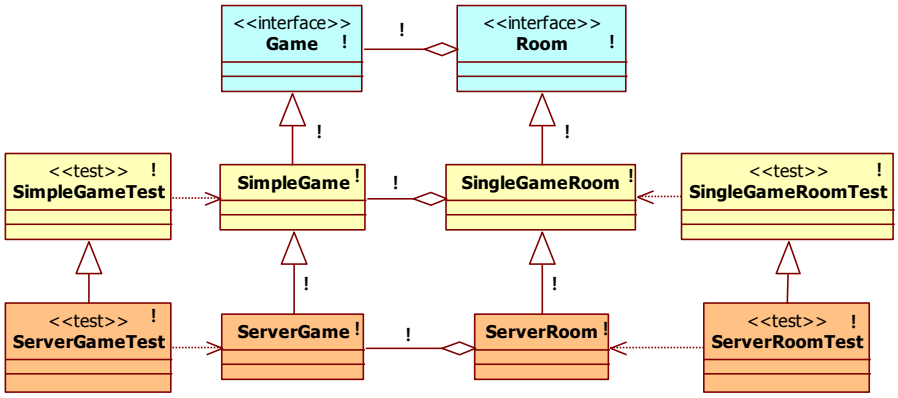


Figure 39. Correctness conditions for interfaces in *ReThink*.

10.6. Inferring correctness conditions

Data refinement and superposition refinement, which are the underlying theory for correctness concerns in SFI, are transitive and monotonic. Therefore, it is possible to infer a number of correct associations without the need to explicitly show them in a diagram. In Figure 40 such associations are marked with a dash line. These inferred correct associations can be of significant use when the system is later extended and new layers are added.

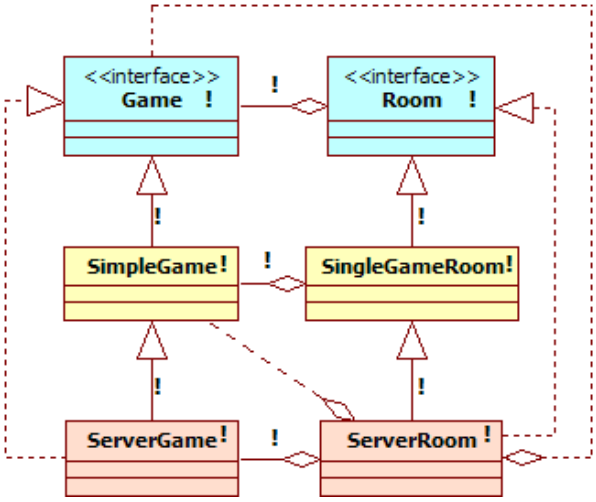


Figure 40. Inferred correctness conditions in *ReThink*.

Part IV:
Case study – BioImageXD2

11. Overview

BioImageXD is free and open-source software for analysis and visualisation of multidimensional biomedical images [20]. The software is written mostly in Python, a free, object-oriented programming language [138], with some code in C++. *BioImageXD* is the result of collaboration between microscopists, cell biologists and programmers from the Universities of Jyväskylä and Turku in Finland, Max Planck Institute CBG in Dresden, Germany and other partners worldwide [20].

The software development project is a part of the national, interdisciplinary science project *BioTarget* funded by the Academy of Finland in years 2007-2010. The goal of the project is to find ways to guide nanoparticles carrying specific toxins to specific cells in human body. Four working groups were formed in this project, roughly corresponding to the scientific disciplines involved.

The material science group (lead by Mika Lindén) focused their research on building nanoparticles that are suitable for both carrying the toxin and for targeting the right cells. The medical science group (under supervision of Sirpa Jalkanen) has focused on developing the toxins and targeting substances for the nanoparticles. The microscope research group (with Jyrki Heino as a leader) supported the material science and the medical science groups by providing microscopic techniques for observing *in vivo* the nanoparticles as they move into the cell and find their target inside the cell. Analysing the results of such experiments provides important feedback to and experimental data for the first two research groups. Finally, the software development group (lead and supervised by Ralph-Johan Back) has been responsible for providing what can be seen as a missing link in this research – the software for analysing the images produced by the microscopes used in the experiments – the *BioImageXD* software.

The researchers use confocal microscopes, in which point illumination and a spatial pinhole are used to eliminate out-of focus light in specimens that are thicker than the focal plane. This imagining technique is used to increase optical resolution and contrast of an image obtained by the microscope, as shown in Figure 41 [185]. Furthermore, it enables reconstruction of three-dimensional structures from obtained images [135].

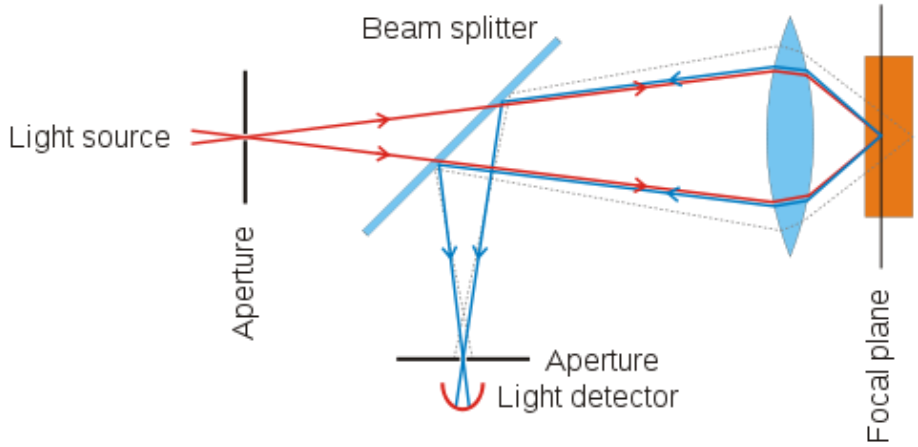


Figure 41. Principle of confocal microscopy.

11.1. First release

BioImageXD has been under the development since approximately 2002 as a side-project to a microscopy-oriented research theme. At the time the development begun, there was no open-source software designed specifically for analysing multi-dimensional images obtained from confocal microscopes. In the later years the software was extended to accommodate functionality needed also by other research projects. These extensions, however, were made *ad hoc*, resulting in gradually degrading system architecture; the software was becoming less reliable over time and difficult to maintain and extend.

The effort needed to modify the system was increasing, so it was decided that restructuring the software was the highest priority for the *BioTarget* project. The software was handed over to the Software Construction Laboratory at Åbo Akademi University in 2007 for improvements and testing. A team of four programmers was formed as part of the internal Gaudí Software Factory and assigned to the project. The goal was to identify key dependencies and structures of the software and to improve them by redesigning the architecture and fixing bugs found during testing. The resources allocated from the project to Software Factory allowed to carry the work during three summer months of 2007.

However, mostly due to underestimating the size of the software and not formulating the goals clearly, none of the above objectives were reached. The inexperience of the development team, combined with the short time allocated, added to the failure of this first attempt at software

restructuring. The software was found to be too complex to handle in a short time: it consisted of more than 75 000 lines of code in Python, without proper documentation. The majority of the time – about two months – was spent on analysing the code and dependencies between different modules, not on fixing bugs as initially expected.

Nonetheless, as the outcome of the three-month project, new long-term goals were set. The new objectives included the lessons learnt during the summer months and the comments of the users of the previous version. We came to the conclusion that the original software had passed the point where it can be easily modified and needs to be refactored or rebuilt. A new architecture must be designed and the existing code base must be adjusted accordingly. Bugs and failures should be captured and eliminated during this process.

11.2. Requirements for refactoring

The goal of the refactoring of *BioImageXD* to its new version, *BioImageXD2*, was to preserve a set of essential features of the original version. These features were selected based on the frequency of use among the users of the first release. Figure 42 lists the customer requirements for the software and divides them into three groups corresponding to the key purposes of the software: accessing, displaying, and processing microscope images.

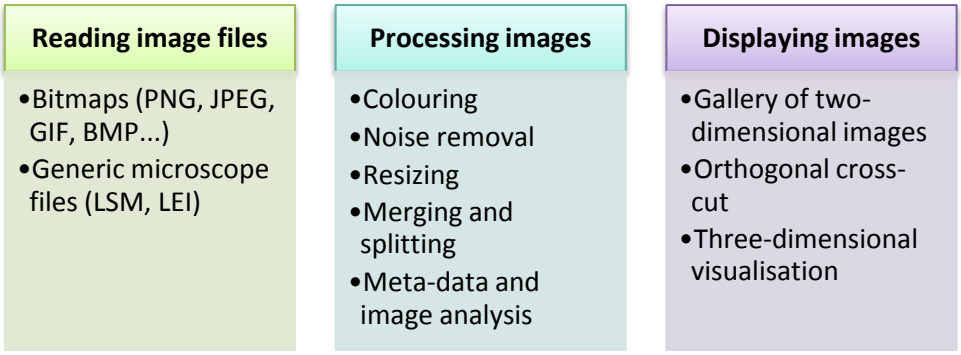


Figure 42. Requirements of *BioImageXD2*.

As previously said, the images are multidimensional. A traditional image has two dimensions – width and height (Figure 43, a)). Third dimension – depth – allows representing real-life objects. Digital microscopes perform a number of two-dimensional scans of the object at certain interval depths,

limited by the resolution. A stack of such images forms a three-dimensional model of the object being viewed with the microscope (Figure 43, b)).

Additionally, the object may be repeatedly scanned with different filters or with light of different wavelengths, similarly to colour components found in e.g. photography (red, green and blue) or publishing (cyan, magenta, yellow and black). Such representations are referred to as channels and form another dimension in the structure of the image (Figure 43, c)).

Performing a number of scans of the complete object in a given time adds the final dimension – time – to the image (Figure 43, d)). The resulting file is thus a collection of two-dimensional images, called *slices*, arranged according to their position in the stack, the channel and the time of their acquisition. We refer to such image as *dataset* to avoid confusion with the common understanding of the term *image*.

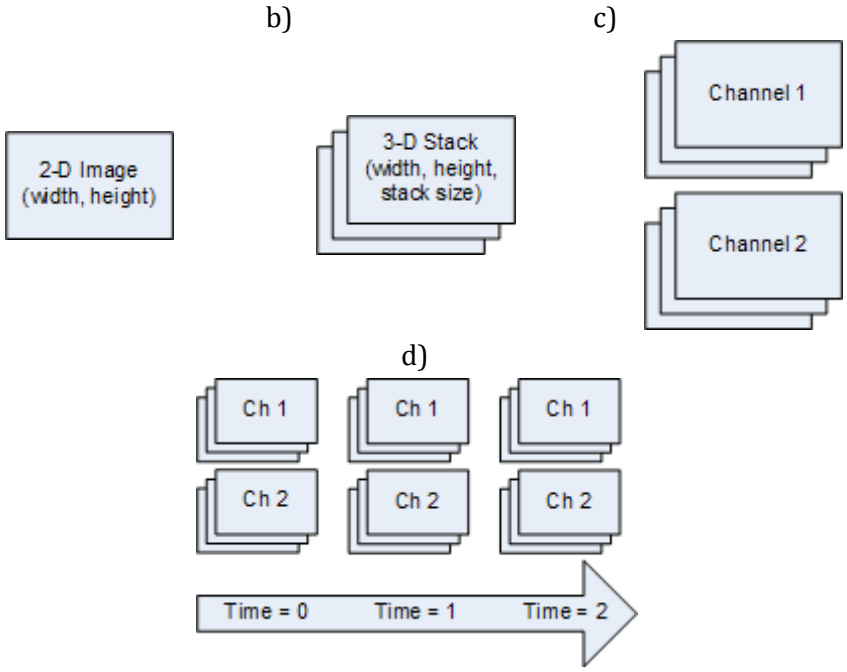


Figure 43. Elements of a dataset.

Once a dataset is acquired from a file, it can be processed or analysed according to certain parameters. Finally, the processed data are displayed to the user. Several visualisation methods are available to give the user better understanding of the multidimensional data.

A typical use case of the software is presented in Figure 44. The process starts with a user opening a file with dataset to be processed.

This dataset would typically have been produced by the confocal microscope as part of some specific experiment. The results of the processing are shown on screen, so that they can be used as feedback. When the processing and analysis are done, the user can save a modified file and the results of the analysis to disk. This use case has a significant influence on the architecture of the software, described in the next section.

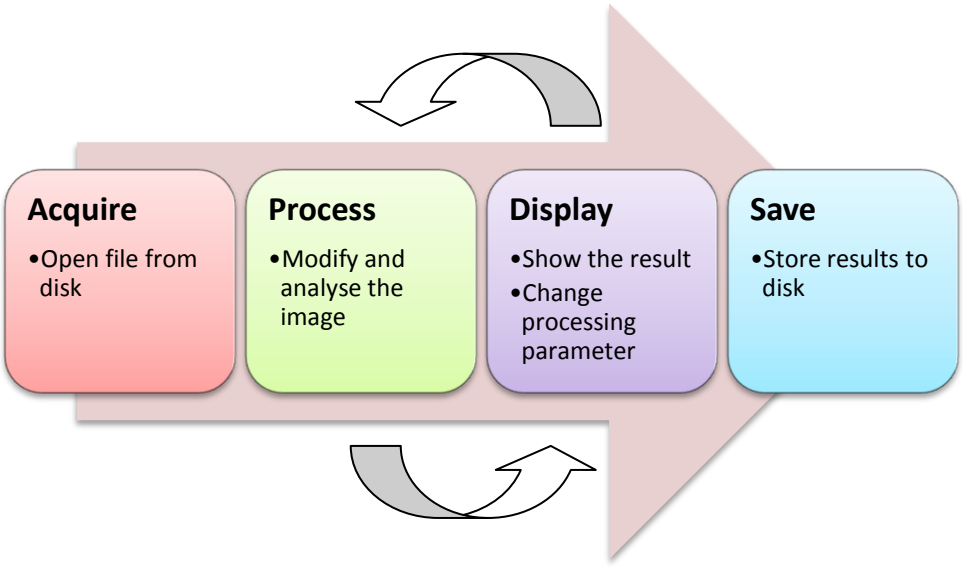


Figure 44. Basic workflow of *BioImageXD2*.

12. System architecture

The analysis of the use case reveals several key parts of the architecture, each corresponding to a step in the scenario. The use case also implies that at runtime the objects derived from these components are chained one to another to enable data flow. In other words, the software must provide means for the user to construct a pipeline of objects created from the components, as shown in Figure 45.

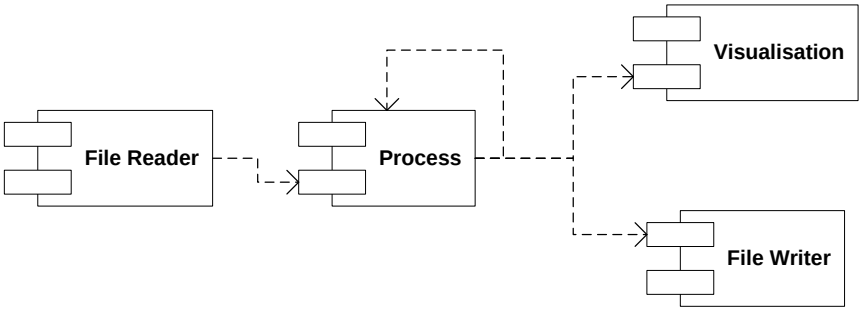


Figure 45. The image processing pipeline.

Pipelining in general refers to a segmentation of a computational process into several sub-processes which are executed by dedicated autonomous units (facilities, pipelining segments). It can also be defined as the technique of decomposing a repeated sequential process into sub-processes, each of which can be executed efficiently on a special dedicated autonomous module [140]. Such definition indirectly requires the segments to be able to take output of other segments as input.

In the context of *BioImageXD*2 the segments of the computational process are the individual processing units. They share the type of input and output, as shown in Table 9. The remaining components are used to feed the data to the pipeline or to handle its output. We will now introduce and describe the components of the system before we discuss them in more details in the next sections.

Component	Input	Output
File Reader	(file)	Dataset
Process	Dataset	Dataset
Visualisation	Dataset	(on-screen display)
File Writer	Dataset	(file)

Table 9. Pipeline of the key components in *BioImageXD2*.

12.1. Representing datasets: BioData

As mentioned previously, the input data may be available in one of many file formats. While the software must be aware of that, its functionality may not depend on the particular format the dataset was acquired from, hence the need for a separate entity for representing the data internally.

The datasets are multidimensional structures composed from two-dimensional images. Each image is identified by its position in the stack of cross-cuts, the channel and the point in time at which it was acquired.

Moreover, each part of the dataset should be considered a dataset as well to enable efficient processing only of a certain part of the set. The UML diagram of the structure is shown in Figure 46.

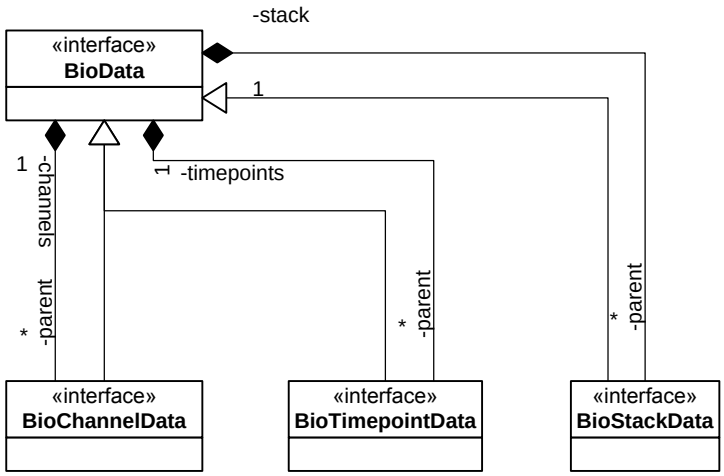


Figure 46. The structure BioData.

`BioData` represents a dataset. It consists of channel-, time-point- and stack-specific data, named `BioChannelData`, `BioTimepointData` and `BioStackData`, respectively. These entities inherit from the interface `BioData`, creating a recursive structure. This design decision allows each dimension-specific data to be also treated as a dataset itself and contain its own channels, time-points and stack of images. Such approach provides an opportunity to later improve the techniques for dataset manipulation, e.g. by adding support for parallel image processing without changing the existing code base.

12.2. Acquiring datasets: File Readers

The File Reader component is responsible for inputting data into the pipeline. Figure 47 lists the relation between three entities used in this process. We decided to make a distinction between a dataset, a file that contains it and a file reader. As a result, each of the classes `FileReader`, `BioFile` and `BioData` has a clearly defined responsibility, in accordance with the principles of object-oriented design.

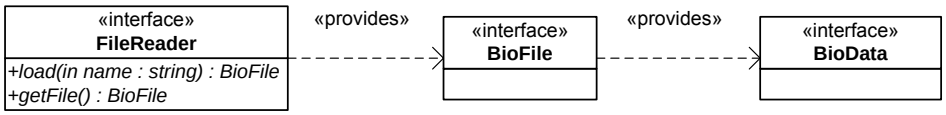


Figure 47. Relation between FileReader, BioFile and BioData.

The sole responsibility of `FileReader` is to load a file from disk. The file is represented by an instance of `BioFile`, which in turn has to create the `BioData` based on the internal format of the file. This approach allows us to use one `FileReader` for a number of different file formats, each represented with a separate subclass of `BioFile`.

12.3. Modification and Analysis: Processes

From the perspective of the customer and the end-users, Processes are the most important component of the system, as they allow modification and analysis of the datasets. The design of the component is shown in Figure 48.

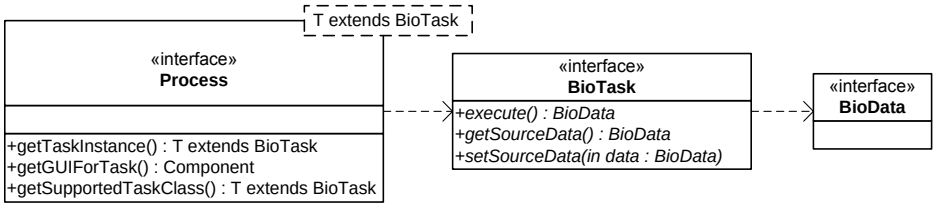


Figure 48. The design of the image-processing component.

We decided to divide the functionality into two separate entities, `Process` and `BioTask`. The former represents the graphical user interface of the latter, which is responsible for the actual analysis and modification of the data. Such an approach not only adheres to the principles of object-oriented design, but also separates the concerns of the graphical user interface design from image processing.

12.4. Displaying: Visualisations

Visualisations (or views) are located at the end of the image processing pipeline to show the end-user the effects of the processing made to the dataset. As the data is multidimensional, different visualisation modes must be enabled to provide full overview of the dataset. The customer requirements for the software included three visualisation modes: gallery, orthogonal and three-dimensional.

The gallery visualisation represents the dataset as a series of two-dimensional images and allows navigating between different dimensions. Each image is shown individually and in a context of its time-point, channel or position in stack, as presented in Figure 49.

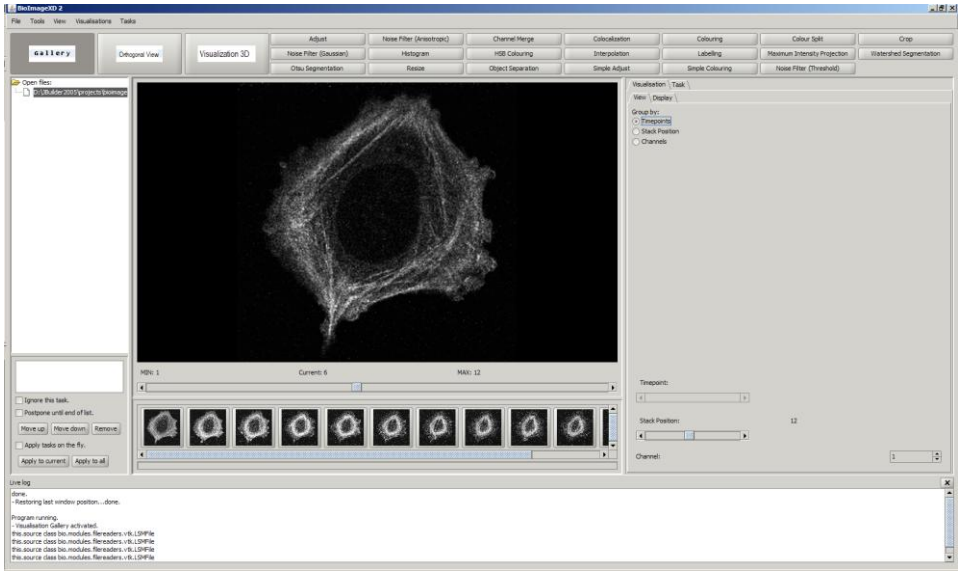


Figure 49. Gallery visualisation in *BioImageXD2*.

The orthogonal view focuses on providing a cross-cut of the stack of images in a given time-point and channel. More precisely, a two-dimensional image is shown together with side views of the entire stack at the positions of the cuts. This visualisation enables careful pixel-by-pixel examination of the dataset. The orthogonal visualisation is shown in Figure 50.

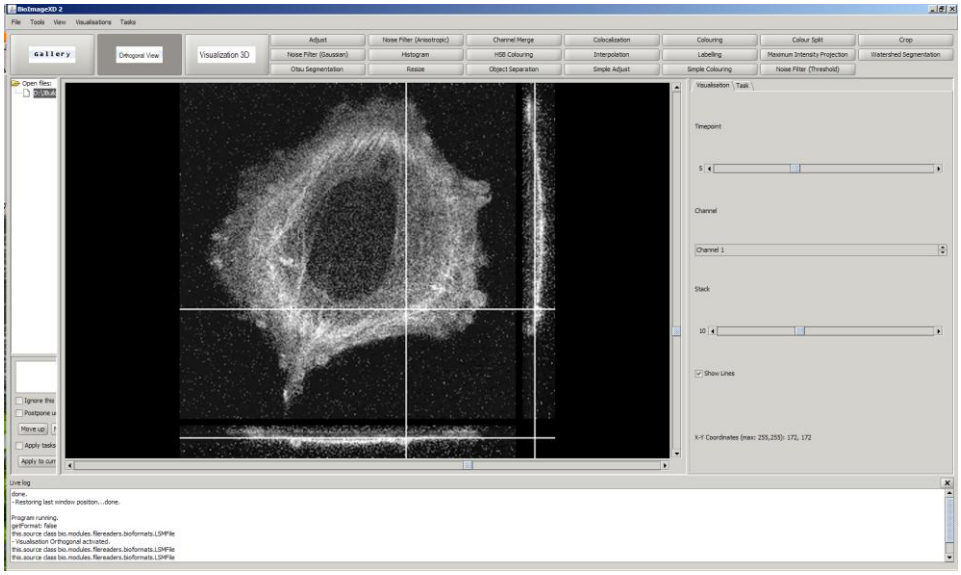


Figure 50. Orthogonal view in *BioImageXD2*.

Finally, the tree-dimensional view renders an entire stack of one or more channels in a given time-point of the dataset. The user is thus given a comprehensive view of the object the dataset represents in a given time point, as presented in Figure 51. The features of the visualisation allow the object to be rotated, zoomed and textured differently, depending on the configuration.

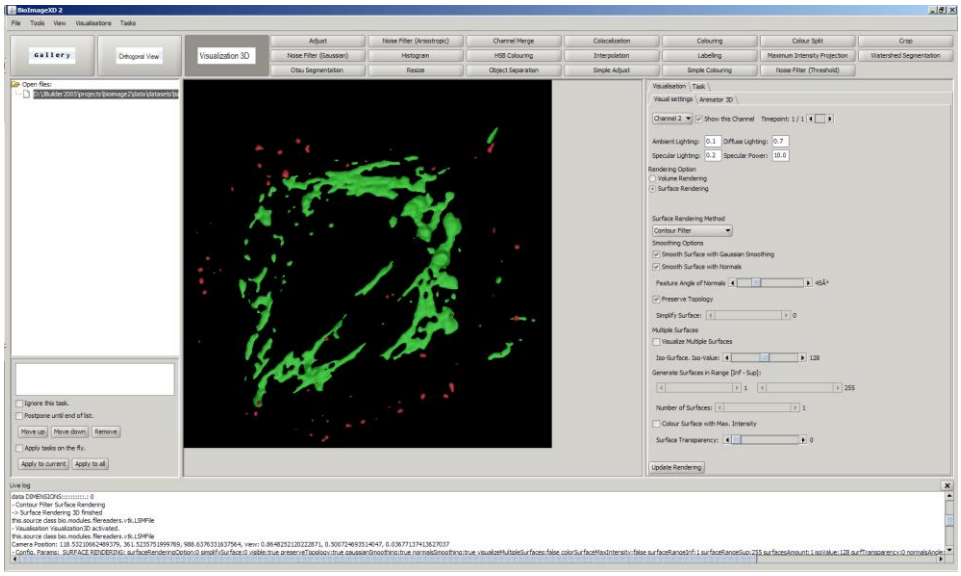


Figure 51. Three-dimensional visualisation in *BioImageXD2*.

12.5. Saving changes: File Writers

File Writers are the components that handle the output of the processing pipeline. The responsibility of a writer is to save the structure and images of the processed dataset to disk. Contrary to the File Readers, File Writers were designed to handle one file format per writer.

12.6. Executable

The components do not contain code or logic related to the execution of the system. The construction of the image-processing pipeline must be done based on the actions of the end-user, thus including the code responsible for that in the components would violate the principles of object-oriented design. To follow the best practices the layers of the components must be executed by a separate entity.

We decided to construct a dedicated system executable that would be extended and modified whenever a need arises. Such executable relies

on the logic of the system (i.e. the image-processing pipeline) and is able to invoke the code in the components at any given layer. Furthermore, the executable is responsible for constructing the graphical user interface and responding to the events triggered by the end user.

12.7. Architectural styles

The architecture of a large and complex software system shares elements of various styles. The most notable characteristic is the layered structure of the system and its components. It is a direct consequence of applying SFI to the development, although other architectural styles can also be observed.

The image-processing pipeline, as the name implies, derives from the pipeline architecture that connects different filters [62]. The filters are in this case the image-processing modules. However, contrary to the pipeline style, the datasets are not processed incrementally and continuously. Not only the processing must be explicitly invoked, but the processes modify the dataset one after another. In other words, entire dataset is passed between different modules once it is processed. On the other hand, the structure of the interface `BioData` enables processing only a selected part of the whole dataset.

Event-based approach [62] is used in the structures related to handling modules (i.e. process, visualisation, file reader or file writer), as well as in connecting the image-processing pipeline to the graphical user interface. This allows decentralisation of the software – its parts operate independently and are invoked implicitly, as a response to certain events in the system. For example, loading a module broadcasts an event. As a response to such event a code responsible for updating the graphical user interface is executed, together with the code that initialises the loaded module.

13. Layered design

The paradigm of Stepwise Feature Introduction supports bottom-up software development. Its application results in software having a layered design. In case of complex systems a reverse approach is often more suitable, as the software must fit the architecture and its bounds. The layered structure, however, is also present.

13.1. File Readers

The sole purpose of a File Reader, as the name implies, is to read a dataset file and represent it as a `BioData` structure to enable processing and visualising. As mentioned previously, we have separated that functionality into three basic concepts. Objects implementing the interface `BioData` represent datasets; descendants of `BioFile` correspond to individual files, whereas `FileReaders` are used to construct `BioFile` from a file physically located on disk. The crucial functionality is thus contained in the descendants of `BioFile`, as these classes deal with the internal structure of dataset files.

During the first iteration two layers are added to the system. One contains mostly interfaces and abstract classes that declare the functionality; the other is a direct implementation of the former, as shown in Figure 52. The essential property of SFI is therefore preserved, since a system that can be executed is produced.

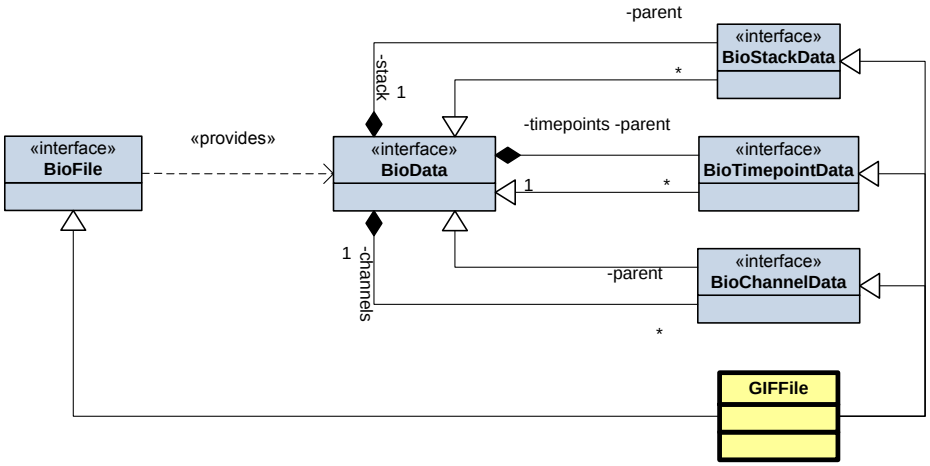


Figure 52. Initial step of introducing features to *BioImageXD2*.

In most cases during the development the new feature is added to the system as another direct implementation of the interfaces (Figure 53, a). The optimisation of the structure follows, in order to remove redundant code and improve performance. As a consequence a previously added layer is updated to contain common code, from which subsequent related features may extend (Figure 53, b). The classes with the common code do not necessarily have to be executable by themselves – they are treated as an integral part of the layer they were abstracted from. The essential ability of the system to be executed is thus still preserved.

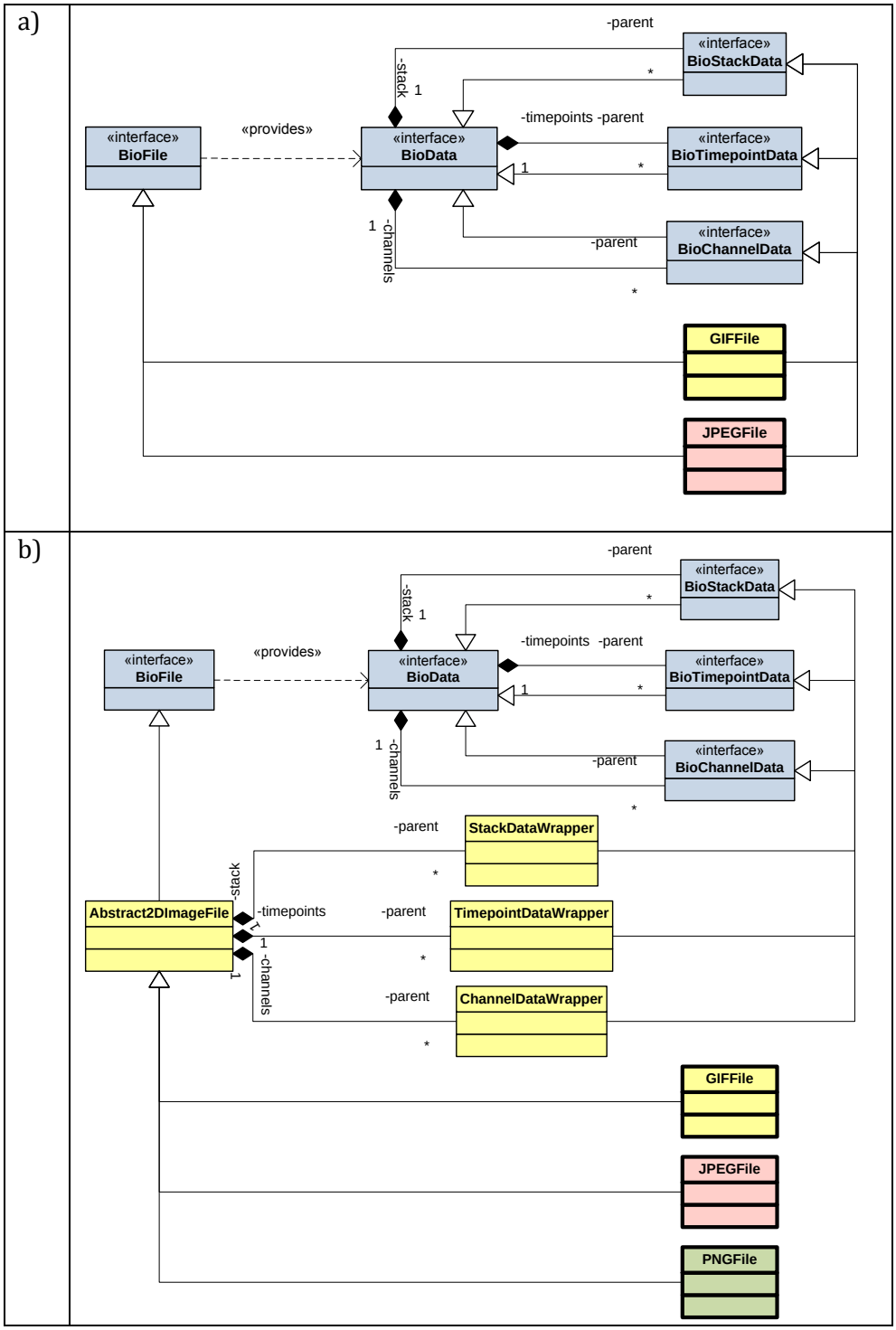


Figure 53. Overview of introducing features to *BioImageXD2*.

The abovementioned layering pattern directly follows from both SFI and the principles of object-oriented design. The pattern is repeated with each new feature added, as seen in the final structure of classes related to `BioFile` and `BioData` shown in Figure 54.

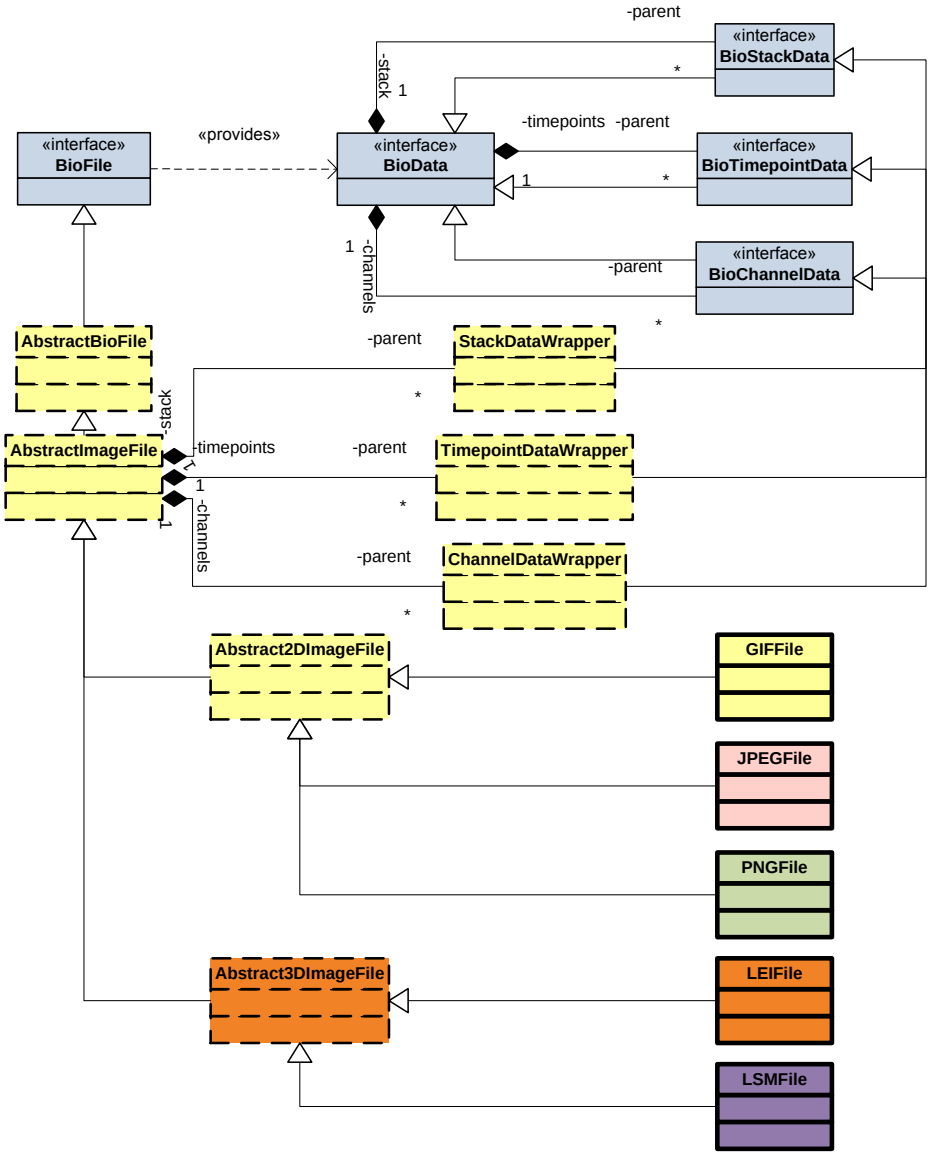


Figure 54. Structures related to file reading.

We can indicate three types of classes that are present in the design. The abstract interfaces (marked with a thin, solid border in Figure 54 and subsequent figures in this part) declare functionality and constitute bounds

for the development. The customer requirements are realised and implemented by the concrete classes (thick, solid border), whereas the code shared among a number of layers is contained in the common classes (dashed border). Thus, based on the component roles in SFI we can state that the common classes are service providers, whereas the concrete classes are users of these services. Since the interfaces set bounds for how each service is implemented, we propose to use the term *service border*. The service borders can also be identified in previously described case study, *ReThink*.

The interaction between the different parts of the system is declared in the interface layer. This can be seen as a potential drawback to the approach, as whenever a new component is introduced to the system, all layers of the system must be updated to accommodate new behaviour. Therefore, it is essential to design an architecture that is easy to extend and stable at the same time to limit the propagation of changes.

An approach similar to the one just presented has been applied to the development of other components of the system. While the number of layers differs from one component to another, the overall organisation of the code is preserved.

A system developed with SFI must be executable after every development iteration. We have sustained this property with the use of a dedicated system executable, which may be configured to ensure that the execution happens at a defined layer. Furthermore, during the development the interfaces were always introduced to the system accompanied by at least one direct implementation, in order for the system to be executable.

13.2. Processes

The functionality related to processing images is delegated to two types: `Process`es responsible for the graphical user interface and `BioTask`s that perform the computations. This separates the graphical user interface from the processing and allows easier use of third-party libraries.

A number of external software libraries for efficient image processing are available. In the field of bio-imaging two such products are commonly used. The Visualisation Toolkit (VTK) is an open-source, freely available software system for 3D computer graphics, image processing and visualisation [81]. The Insight Segmentation and Registration Toolkit (ITK) is a cross-platform system that provides developers with a suite of software tools for image analysis [80]. To enable more efficient processing of the

images the support for at least these third-party libraries must be added to the system.

As with other components, the direct implementations were added first. Subsequent layers were inserted as a result of code optimisation in the next cycles. The support for third-party image-processing libraries is encapsulated in a separate layer that provides an entry-point for future enhancements. The extended structure is shown in Figure 55.

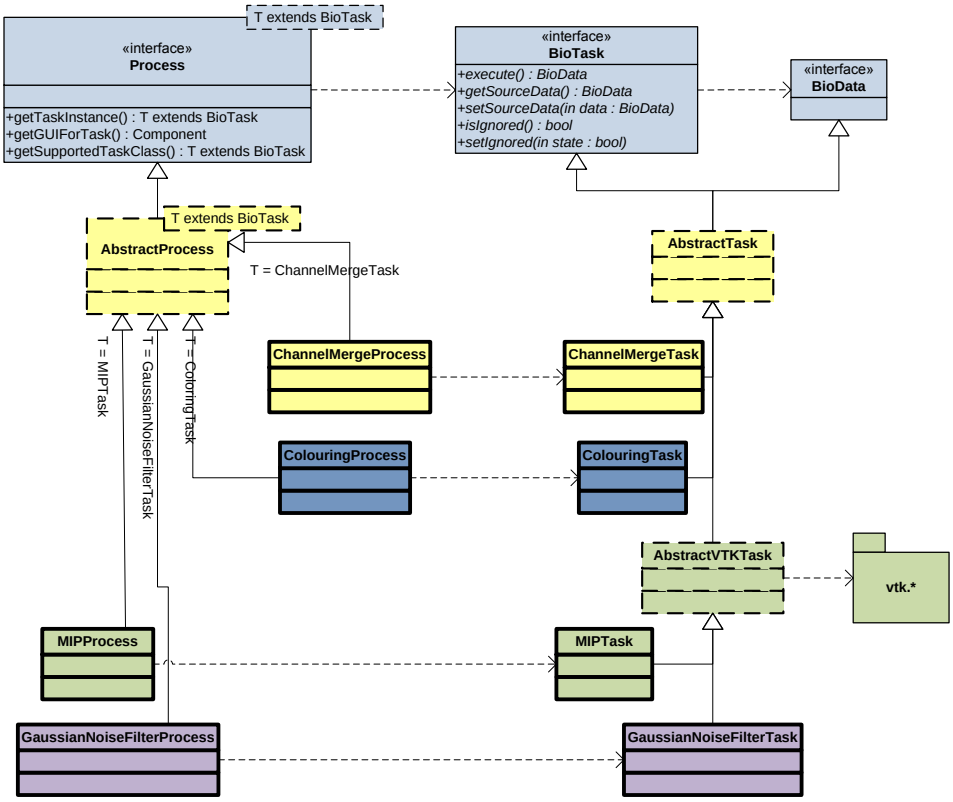


Figure 55. Structures related to image processing with external library support.

An important observation should be made at this point. The software project focused on re-engineering an existing version of *BioImageXD*. Therefore, the major concepts and the final functionality of the overall system (i.e. the interfaces and the concrete classes) were fixed and could not be modified. SFI was used to optimise the structure of the code and, through careful introduction of functionality, increase the quality of the system.

13.3. Visualisations

Visualisations are the final components in the image-processing pipeline. They are used solely for displaying the image, so that the user can instantly see the outcome of processing and adjust its parameters as needed. The classes related to visualisation are shown in Figure 56. The presented structure is straightforward and follows the layering pattern described earlier in this chapter. The low number of classes that contain common code is caused by significant differences between the visualisations.

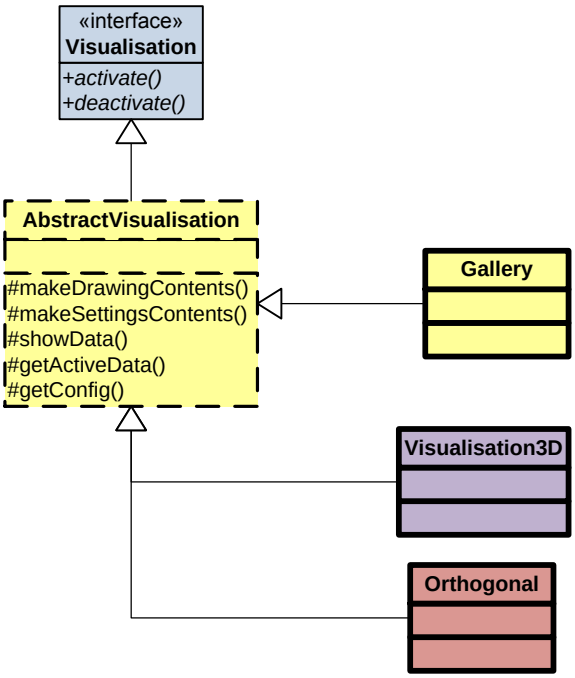


Figure 56. Structures related to image visualisation.

13.4. File writers

File writers, in terms of functionality, are similar to file readers, as they operate on files and different image formats. However, with respect to their role in the system, the writers bear more resemblance to visualisations, as they function at the end of the image processing pipeline. Contrary to image visualisations, the writers do not serve as the feedback to the end users. Instead, they allow the results of the current processing to be saved and reopened later.

The structure and layering of the file writers is shown in Figure 57. Due to differences between various file formats the optimisation of the

structure – and hence the insertion of the intermediate layers – was not necessary.

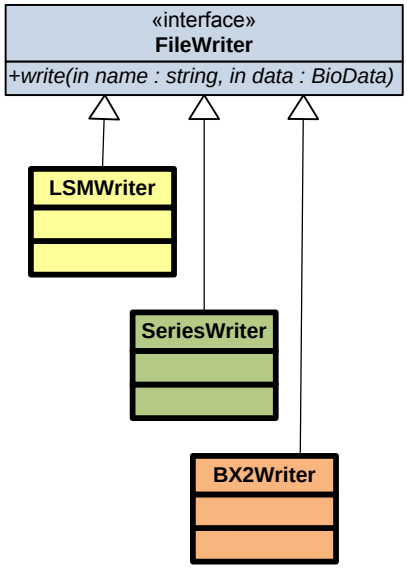


Figure 57. Classes related to file writers.

13.5. Modularisation

The original version of *BioImageXD* offered its users a possibility to extend the core functionality with custom scripts written in Python. One of the most important goals set for the re-engineering process was to preserve and improve this functionality. In particular, it should be possible for the users to provide additional file readers, file writers, visualisation modes and, most importantly, processes, through plug-ins (modules).

To ensure that the architecture supports future extensions we decided to design the software so that all the core functionality itself is contained in different plug-ins. Introducing modules, however, was postponed until all kinds of potential modules had at least one direct implementation. Such approach allowed us to identify the commonalities between different kinds of components and define the mechanism form handling them, as shown in Figure 58.

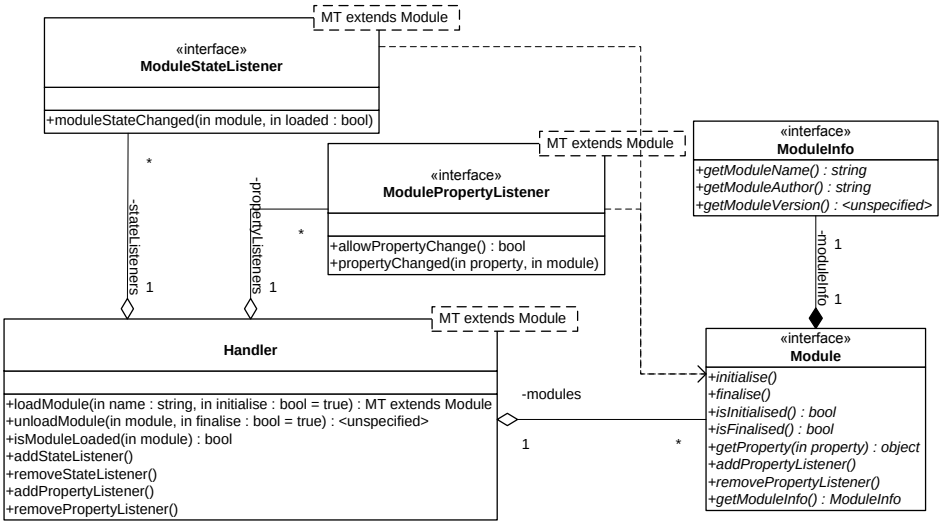


Figure 58. Components responsible for modularisation.

The functionality common to all kinds of plug-ins is encapsulated in the interface `Module`. This interface defines a way for a module to initialise and finalise its resources. Furthermore, it provides a method for other objects to obtain module properties. A class for storing essential module information (its name, author and version number) is also defined.

Each subclass of module constitutes a separate type that contains additional behaviour. This additional functionality does not rely on the architecture of the software. However, to enable new functions each module must be properly loaded when the software starts. Class `Handler` provides methods for loading and unloading modules, regardless of their type.

The Observer pattern [61] is applied to both `Module` and `Handler` to notify interested entities about changes to the properties. The former broadcasts events whenever an internal property is about to be changed and when such change occurs, whereas the latter informs upon successful loading or unloading of a module.

The interfaces `Module` and `Handler` were inserted to the system as its first, abstract layer. As a consequence, the subsequent layers had to be updated to include the changes, as shown in Figure 59. The basic components of the system (file readers, file writers, visualisations and processes) were modified to extend the interface Module. Moreover, subclasses of `Handler` specific for each type of module were created.

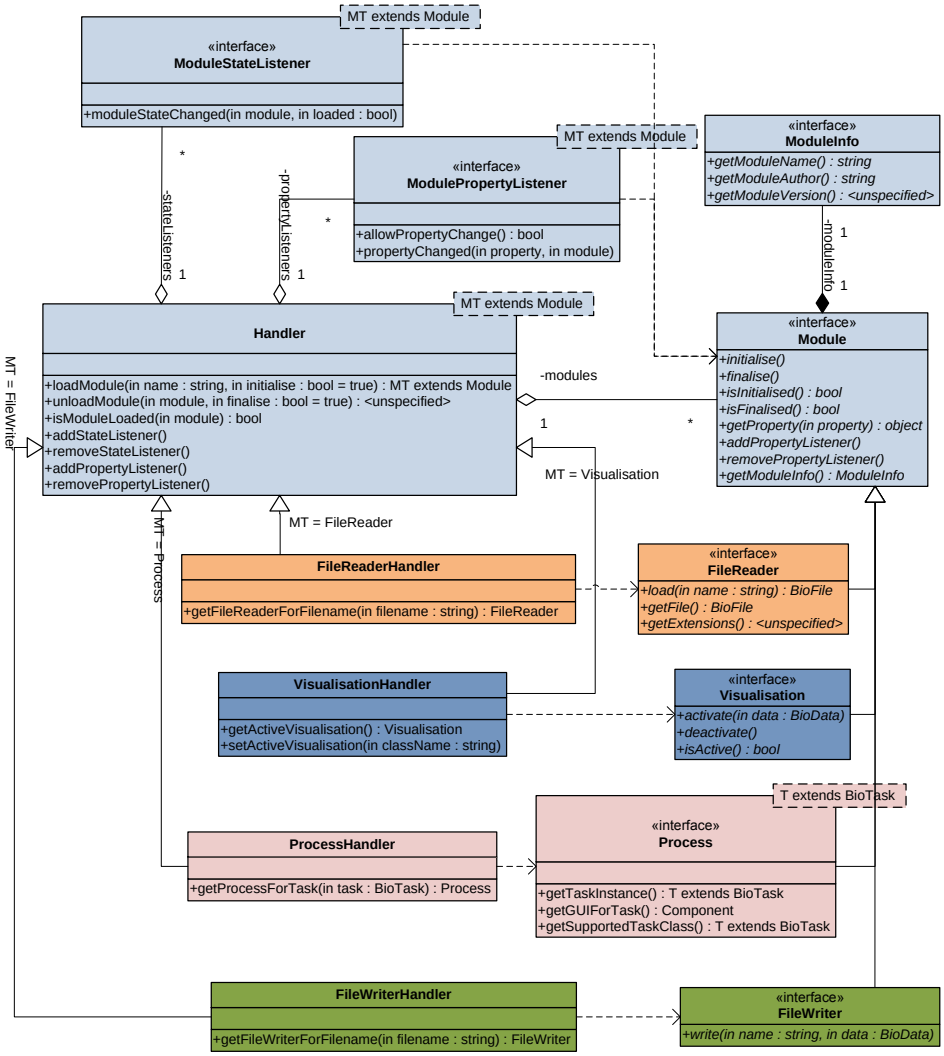


Figure 59. The connection between the module defining layer and the basic components.

The abovementioned structures for handling modules of different type were included in the system executable. This allowed each layer of any component of the system to be executed independently. The changes reduced the role of the system executable to loading available modules of different kinds and providing a graphical front-end for the user.

Each plug-in is expected to be released and packaged separately from others, to further underline the independence between modules. Furthermore, the separation between different kinds of modules allows the

classes that implement the requirements to be executed in separation from the rest of the system.

The layered structure presented in Figure 59 shows also the order in which the components were introduced to the system. This order corresponds to the image processing pipeline. The file readers were constructed first, in order to be able to read images. Subsequently, the visualisations were designed and introduced to examine whether the file has been read correctly. These two components received their initial implementations in parallel. Finally, the processes and file writers were added. This approach to the development gave more control over the graphical user interface and the overall functionality of the system.

13.6. Testing

The system was constantly tested during its development in order to increase its quality. Different kinds of testing strategies were utilised, depending on the suitability to the development problem. In addition to regular tests, the source code was frequently reviewed and refactored. The corrections and revisions were communicated back to the development team in order to prevent repeating coding errors and mistakes in the future.

Unit testing was applied to ensure that the code functions properly. During the implementation of the system we followed the guidelines of Test-Drive Development [15] to verify that the most complex and demanding situations are handled as expected. Furthermore, the unit tests were used to cover regression tests in order to satisfy the correctness conditions of SFI, as described earlier in the thesis.

The strategy to unit testing differed between the components. The file readers were tested whether they are able to correctly represent the structure of those files, for which the properties were known by the development team in advance. The test data included two- and multi-dimensional datasets to cover most of the real-life situations.

The file readers were also utilised in the testing of file writers. Each test consists of three operations performed on the same dataset. At first the dataset is read by an already tested file reader in order to obtain a `BioData` structure. This structure is then saved by a file writer and a subsequent reading of the written file is performed in order to compare its contents with the original one.

A different approach was applied to the testing of the processes. A number of properties shared across different processes were identified.

Subsequently, a parameterised test case for each property was constructed. As a consequence, every process was tested by a number of such test cases.

The unit testing of visualisations was limited to ensuring that their internal configuration works as expected. This restriction was caused by the fact that visualisations are modules oriented towards graphical user interface and, as such, their unit testing is more difficult [112], than in case of other parts of the system.

14. Development process

The development process of *BioImageXD2* can be seen from two perspectives. In the large scale – the reengineering of a complete system to fit its new architecture – prototyping was used. The organisation of the work on a daily basis, however, was heavily influenced by agile development methods.

14.1. Prototype development

Software prototyping is an activity of creating incomplete versions (prototypes) of the system before it is fully developed. A prototype typically simulates only a few aspects of the features of the final system and may be completely different from its implementation [186].

The process of prototyping is divided into four stages [118]. At first, the basic requirements of the system must be identified. The emphasis is to recognise the essential features of the system and the relations between them, as well as the basic elements of the development process. It is important to notice that by no means should the list of identified requirements be complete, as only the most important ones must be defined.

Development of a working prototype is a second step in the process. The prototype should ideally be implemented in a short time, as to increase benefits of both the users and the developers. The former receive a tangible system to experience and criticise, the latter a basis of the development process to improve further.

The prototyping follows with hands-on use of the system and its evaluation. The feedback from the end users enables the developers to enhance the final version of the system. At the same time the development

process should be reviewed to ensure that the final step in the prototyping is carried out efficiently.

Enhancing of the prototype – correcting the flaws identified during the previous step and adding missing features – ends the process. The revised prototype may form a complete system, or may be given to its users for additional evaluation.

Two distinct approaches of prototyping are commonly used: throwaway and evolutionary prototyping, both reflecting their names. The latter depends on gradual improvements to the prototype according to the suggestions and requirements of the users. This iterative process finally leads to the construction of the complete system.

Throwaway prototyping, on the other hand, discards the prototype once it is evaluated. This means that the first model of the system does not serve as a basis of the final system. This decision is usually supported by the rapid development of the prototype and the fact that it only simulates parts of the final functionality. Constructing a complete system without the code of the prototype ensures that the developers focus on a good design.

Between September 2007 and May 2008 the architecture of the software has been defined, as described in the previous sections. For the summer of 2008 a Gaudí Software Factory team was formed to construct a prototype of the project. The team included four programmers and was given three months to complete a prototype based on a given architecture and a limited number of features from the original version of the software.

14.2. Evaluation of the prototype

We decided to follow the throwaway prototype development model. A limited version of the software was produced first to test and evaluate the key assumptions of the architecture. The lessons learnt were used to improve the process, adjust the goals and enhance the design during the second phase, when a working version of the software was produced.

As a result, the architecture was refined to better suit the software. Moreover, three major concerns over the development were identified. First, the programming language seemed to be an obstacle in the development, mainly because of the size of the software. Second, the team found the used development methodology inappropriate. And third, the motivation of the programming team was a problem during the fixed three-month period.

The issue of motivation of the team members was solved by reducing the number of programmers in favour of extending the time allocated for the project. Thus, the implementation of the final system was performed by three programmers in the period of six months, later extended to a year. The two remaining issues required more sophisticated solutions.

Programming language

The first version of the software was written using Python, an interpreted, dynamically-typed, object-oriented programming language [138]. The original release contained about 75000 lines of code. The lack of adequate documentation and *ad hoc* architecture negatively influenced the possibility of analysing the code. Further obstacles were caused by the dynamic typing and, in consequence, the lack of type information in the code.

On the other hand, Python is known to have a low learning curve [49]. It is also easy to understand, especially for people without prior experience in programming. As the programmers involved in the project were students, these advantages seemed more important than the drawbacks, as compared in Figure 60.

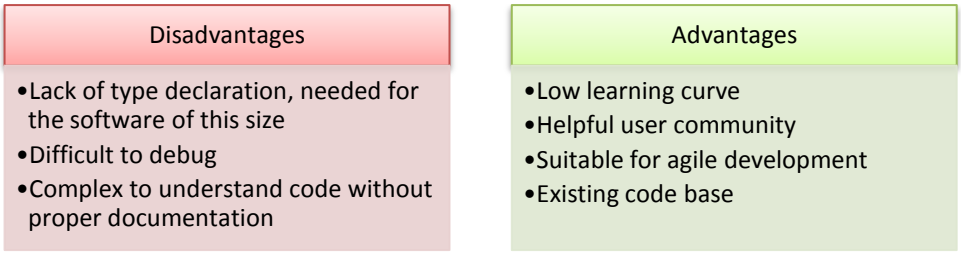


Figure 60. Advantages and disadvantages of Python in the original version of *BioImageXD*.

During the development of the prototype it became evident that Python – a dynamic, interpreted language – is not suitable for a construction of such a complex software system. Several discussions with the stakeholders lead to the unanimous decision to switch to Java [132]. This decision was made despite some well-known and widely discussed issues [187], including performance and memory management. The former, however, in the most recent Java versions is not a significant drawback, as the virtual machine executes the code with speed comparable to compiled languages [129]. Effective memory management can be addressed by enforcing a programming discipline that limits the number of objects present in memory at any given time.

This change was further supported with strong arguments. Java is a multi-platform language, which means that the software created with it can operate under different operating systems. As opposed to Python, Java is a compiled language with static typing, which greatly increases the understanding of the source code. Moreover, it is a commonly known and well-established language, with a large number of libraries that provide additional functionality. Finally, the programmers were familiar with Java and programming techniques in general, so that the programming could continue in a different language.

A major difference that needed to be tackled during redesigning was the difference in inheritance mechanisms between the languages. Python supports multiple inheritance, whereas Java allows single inheritance with multiple interfaces. Therefore, the design of *BioImageXD2* relies more on composition than inheritance between concrete classes.

Development methodology

The development process used during prototyping was also investigated. We used Extreme Programming [14] mainly due to successful application of this paradigm to the past projects [9][116]. Unexpectedly, the team members did not feel content with the process. They reportedly felt unnecessary pressure from pair programming, which is considered an essential feature of XP. Frequent discussions with the team lead to abandoning pair programming, unless needed for the most complex tasks.

As a result, we decided to use a different process for the development of the final version of the system. We created a Scrum-based process integrated with Stepwise Feature Introduction, which we presented earlier. To further improve the quality of the software we decided to organise two additional meetings held at the beginning of each working day, after the daily meeting. These meetings were neither obligatory nor time-boxed, although the maximum recommended time was one hour for both of them. They were organised according to current needs. The first meeting focused on the code produced by the programmers and the results of code inspections. The other meeting was intended to be an open discussion among Team members and concerned the design of the system.

The process gave the team the freedom to organise their work and to decide whether or not use pair programming. Moreover, the daily meetings enforced an organised way to ensure that the development follows the plan and both the design and the code are of good quality.

Part V:
Evaluation

15. Evaluation criteria

We expect that the application of Stepwise Feature Introduction to the development of software systems brings significant benefits to the constructed system. The paradigm should also help in establishing a maintainable and reusable structure of the produced software. In consequence, the overall quality of the developed system should be high. Due to its underlying principles, SFI ought to bring a high level of rigour, which is particularly beneficial when using agile development processes. Finally, as SFI relies on object-orientation, the common principles of object-oriented design must be preserved. All of the above properties should be true not only for the paradigm in its originally proposed form [8], but also for its improved version, presented in this thesis.

The original version of SFI had been successfully applied to the construction of software system of moderate complexity and size [9]. However, in order to evaluate the scaled version of the paradigm and claim whether or not it suits and supports construction of high quality software systems, we need to establish evaluation mechanisms. To validate our claims we rely on a measurement plan used throughout the development of a large-scale software system. Furthermore, we check the software for compliance with the core principles of object-oriented design, which are said to increase the overall quality of the design and the software [109]. Lastly, to gain more data and increase the value of our findings we also interview the developers that worked on one of our case examples.

15.1. Indicators of quality

There is no precise description of what *good* software is, as the quality of software depends on a variety of aspects [55]. Despite this lack of definition, certain characteristics of software systems are considered desirable.

Good software system must perform its functions reliably, without faults and errors. In object-oriented systems this property can be ensured in various ways, depending on allocated resources, development practices and accurateness. On one end of the spectrum there are *Formal Methods,* mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems [29]. Formal verification is expensive when applied extensively or for the first time. However, for high integrity systems such an investment can be warranted and the

returns are usually sufficient to justify the cost [26]. Several formal verification techniques exist for object-oriented systems, e.g. a technique based on object invariants [11][87], which accommodates subclassing and object composition and allows full specification of object-oriented systems. Tests, and in particular Test-Driven Development [15], can be seen as the opposite of formal approaches, as testing does not mathematically guarantee the conformance of the code to the specification. However, a carefully designed test suite consumes fewer resources than formal verification and can be performed by the development team without mathematical background. Finally, code reviews and demonstrations can further contribute to the quality and reliability of the produced software system, but are less accurate than tests.

The second important characteristic of good software is for its architecture to be both adaptable and resilient to change [61]. In other words, the system must be prepared to handle changing requirements and additional functionality without compromising efficiency in performing its current tasks. In addition, good architecture needs to promote reusing parts of the software in other systems operating in related domains.

Maintainability is another important characteristic and indicator of good software. We expect not only the constructed product to be easy to use, but also its source code to be modifiable, readable and well documented. The community of researchers and practitioners have established a number of principles and solutions, adhering to which contributes to reusable, adaptable, maintainable and stable design [61].

Above all, however, the software system must perform what it was designed to. There is little use for a system that operates reliably, can be adapted to different settings and is easily maintainable, if it does not conform to the requirements of its users. In other words, it is not sufficient for the system to *do things right* – it also needs to *do the right things*.

15.2. Quality attributes

ISO/IEC 9126-1 *Software engineering – Product quality* [75] is an international standard for evaluation of software quality. The quality model established in the standard classifies software quality in a structured set of characteristics (functionality, reliability, usability, efficiency, maintainability and portability) and related sub-characteristics [184][75]. Of these main characteristics, in our opinion two are especially important in the context of large and complex software systems: maintainability and usability.

Maintainability is a characteristic that allows drawing conclusions about how well the software can be maintained. Its sub-characteristics, according to the standard, are: analysability, changeability, stability, and testability [75].

Usability is a general term that refers to how well the software can be understood, learned, used and "liked" by the developers. It can be used for assessing, controlling and predicting the extent to which the software product (or parts of it) satisfy usability requirements. The sub-characteristics related to usability follow straightforwardly from the definition. They are: understandability, learnability, operability, and attractiveness [75].

Reusability of software (system, component) is a special case of usability and shares its sub-characteristics. It can be understood as an ability of software for integration in other systems. While we want to ensure maintainability and usability of a software system as a whole, we want its parts (e.g. modules or components) to be reusable.

Furthermore, we are interested in *adaptability*, a sub-characteristic of portability. Adaptability allows drawing conclusions about how well software can be adapted to environmental change [75]. This sub-characteristic is important given the open-endedness of SFI.

15.3. Software metrics

Objective and quantitative methods are needed to state that a system shows certain characteristics. Performing tasks according to the specification and requirements (i.e. *doing things right*) can be achieved through mathematical proofs or successfully completed test suites. Matching the expectations of the end users (i.e. *doing the right things*) can be ensured with a dedicated development process that enables frequent communication between the stakeholders and the developers and allows the latter group to present executable, working versions of the system. Finally, maintainability and reusability, among various other characteristics, can be measured and controlled based on metrics and measurements [55][40] throughout the development, even in its early stages and in rigorous settings [128].

Quality of the design

Several metrics for object-oriented design have been established. Three of them – Abstractness, Instability and Distance – serve as an indicator of design quality and provide information to the designers regarding the

ability of their design to be changed or reused [97]. These metrics do not depend on the source code; therefore the quality of implementation is not taken into account.

Instability metric depends on package coupling. It is defined as a ratio between the outgoing dependencies and the total number of dependencies of a package. The result thus ranges from zero (no outgoing dependencies, stable package) to one (only outgoing dependencies, instable package).

Abstractness of a package is the proportion between the number of contained abstract units (i.e. interfaces or classes declared abstract) and the total number of contained entities. As with Instability, the result is between zero (concrete package) and one (fully abstract package).

The principles of object-oriented design state that the abstraction of a package should be in proportion to its stability. The Distance metric, defined as an absolute value of the sum of Abstractness and Instability minus one, defines the relationship between the stability and abstractness of a package. Packages with Distance close to zero are equally stable and abstract. The metric is also commonly used to identify packages that are either both stable and concrete or instable and abstract, i.e. not changeable or reusable.

Quality and complexity of the implementation

The metrics introduced by Chidamber and Kemerer [40] are the first empirically validated set of metrics designed solely for object-oriented systems. They are commonly used and provide an insight on the complexity, maintainability and understandability of the system without the need to investigate the implementation. Most importantly, measures based on those metrics can be obtained automatically.

The Weighted Method Count (WMC) metric sums methods defined in a class based on a weighting factor. Typically McCabe Complexity [110] metric is used as such factor, resulting in WMC indicating the overall complexity of a class. The high value of WMC negatively affects the maintainability of the class, as it signifies greater complexity and thus more effort to modify the code. Moreover, WMC has an inverse dependency correlation with reusability. High WMC indicates that a class is concrete and, probably, application specific, therefore the possibilities for reuse are limited [40][5][76].

The Depth of Inheritance Tree (DIT) measures the number of ancestors of a class. High value of DIT negatively affects most of the quality attributes related to maintainability and reusability. A class located deep in the hierarchy contains a high number of inherited methods, i.e. the functionality that is not explicitly declared or defined in the class. As a consequence more effort is needed to predict the behaviour of the class, understand or modify it; therefore understandability, learnability, and maintainability decrease with the increase of DIT [5][40].

The Number of Children (NOC) metric is a count of direct subclasses derived from a given class. Since inheritance is a form of reuse, high NOC positively affects reusability. However, changeability – a sub-characteristic of maintainability – decreases with higher NOC, as the changes to the class propagate to all its descendants and it is harder to predict the effects [40][5][76]. Furthermore, if a class has a large number of children it may indicate misuse of sub-classing [40].

Coupling Between Objects (CBO) is a count of the number of other classes to which a given class is coupled. Two classes are coupled if methods of one class use methods or instance variables defined in another class. Excessive coupling prevents reusing classes in different contexts. Furthermore, maintainability is also decreased by high coupling, as analysability and changeability are negatively affected by a higher number of dependencies [40][5].

The Response for a Class (RFC) measures the count of methods that can potentially be executed in response to a message received by that class. The response set for the class are thus the public methods and methods directly called by them. High response decreases maintainability, as it negatively influences all its sub-characteristics. The time and effort needed to understand, analyse or modify the class is increased with high RFC, as there are many method calls involved in a single response. Furthermore, testability is also more difficult, as the number of methods to be tested is greater [40][5].

The Lack of Cohesion in Methods (LCOM) relies on a concept of method similarity. Two methods of the same class are similar if they access a common field. LCOM for a class is calculated as the number of method pairs which are not related minus the number of related pairs; or 0 if such result is negative [126]. Cohesiveness of methods within a class is desirable, as it promotes encapsulation; lack of cohesion implies that a class should probably be divided into two or more subclasses [40]. High LCOM

negatively influences maintainability, portability and reusability, as it is a sign of increased, unnecessary complexity and poor design [5][40].

The effects of the abovementioned six metrics on adaptability and the attributes of maintainability and reusability are given in Table 10 [5]. A minus sign ('-') is used whenever an increase in metric value decreases the attribute; by analogy plus symbol ('+') indicates a positive correlation and zero ('0') shows no effect. For example, the attractiveness increases with the depth of inheritance tree (DIT), but decreases when there is high coupling between objects (CBO). The latter metric, on the other hand, has no effect on learnability and operability.

	Attribute	WMC	DIT	NOC	CBO	RFC	LCOM
Maintain-ability	Analysability	-	-	0	-	-	-
	Changeability	-	-	-	-	-	-
	Stability	0	-	0	-	0	-
	Testability	-	0	0	-	-	-
(Re-) Usability	Understandability	-	-	+	-	-	-
	Learnability	-	-	+	0	0	0
	Operability	-	-	+	0	0	0
	Attractiveness	+	+	+	-	+	-
	Adaptability	-	-	0	-	-	-

Table 10. Effect of Chidamber and Kemerer metrics on quality attributes.

We expect system constructed with SFI to be highly maintainable and adaptable. Moreover, we want the layers located high in the hierarchy or introduced early to the system to be reusable. Based on the effect of metrics on the quality attributes the NOC metric should have high values, whereas other values should be kept low.

16. Empirical validation

BioImageXD2 was the first project built with Stepwise Feature Introduction that any measurement plan was applied to. Thus, with respect to quality assessment, we cannot perform any comparison with other software built with SFI. Moreover, it is also impossible to prove that the paradigm improves the quality of the constructed software compared to systems developed without it. Instead, we may relate our results to generally accepted standards and measure only the overall quality of the constructed software.

16.1. Measurements

The construction of any large and complex software system is tedious and laborious. A discipline to follow the principles of the paradigm is required from the development team and the customer. In order to objectively assess the quality of the constructed system a measurement plan can be prepared and implemented throughout the development.

The measurement plan was created and carried out during and after the development of *BioImageXD2*, using Structure Analysis for Java (STAN) [125] toolset and plug-in Metrics [152] for Eclipse development environment [50]. During the construction of the plan we used the Software Quality Metrics compendium [5] for metrics definitions and quality attributes. The aim of metrics plan is to ensure that the software has met its primary goals, i.e. it is easy to extend and manage.

To enable comprehensive measurements the source code of *BioImageXD2* has been divided into a number of packages, as shown in Figure 61. The division has been made to promote encapsulation of related functionality and to minimise unnecessary coupling. It also reflects the roles of different classes in the system and identifies its parts.

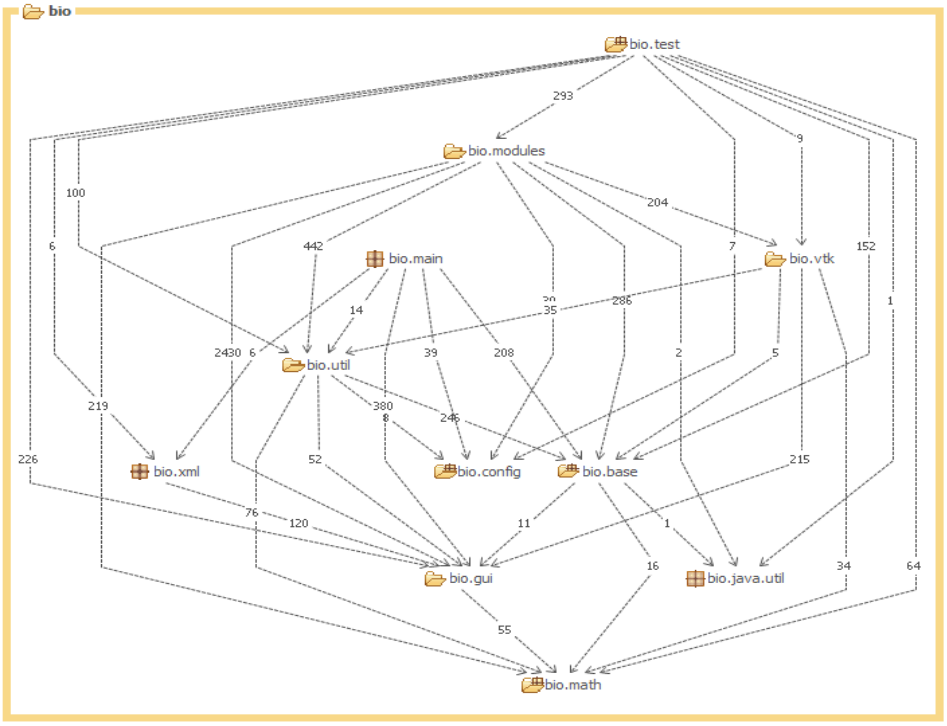


Figure 61. Organisation of the source code of *BioImageXD2*.

Package metrics

Due to the way *BioImageXD2* was constructed, we expect from the architecture that the packages containing the definitions of module types (the abstract layer) are highly abstract and stable. In case of the module implementations we want the opposite to be true: the packages should be very concrete, but instable due to depending on high number of classes. The relation of stability and abstractness should be preserved for the packages that contain the functionality of the intermediate layers.

The results of measurements are given in Table 11. The results follow our predictions. Abstractness decreases while going to the concrete layers, while Instability increases accordingly. Furthermore, all the packages have a reasonable Distance value, meaning that their abstractness and stability are balanced.

Layer	Package	Instability	Abstractness	Distance
Abstract	bio.base.modules	0.2	0.9	0.1
Module handler	bio.base.handlers	0.83	0	0.17
Common implementation (system)	bio.base.util	0.24	0.57	0.19
Common implementation (module)	bio.util.modules	0.45	0.53	0.02
Module implementation	bio.modules. filereaders	1	0	0
	bio.modules. .filewriters	1	0	0
	bio.modules. .processes	1	0	0
	bio.modules. .visualisations	1	0	0

Table 11. Package metrics for modules.

Based on these results we can conclude that the organisation of the source code and its higher-level design are reusable and maintainable. In addition, modular architecture and high abstractness of the component layers increase adaptability, as introducing new functionality is possible without additional effort.

Design complexity

The data for six metrics defined by Chidamber and Kemerer [40] have been collected for the whole system. We present the results based on two examples: processes in Table 12 and file data in Table 13. The former represents modules, while the latter – module-specific implementation of data representation.

Class (layer)	WMC	DIT	NOC	CBO	RFC	LCOM
`Module` (abstract)	0	0	4	8	8	0
`AbstractModule` (common implementation – system)	19	1	5	1	18	51
`AbstractProcess` (common implementation – module)	8	2	10	2	16	15
`AbstractVTKProcess` (common implementation – third-party specific)	8	3	13	0	7	1
`GaussianNoiseFilterProcess` (module implementation)	5	4	0	7	17	4
`ColouringProcess` (module implementation)	55	4	0	27	105	33

Table 12. Results of Chidamber and Kemerer metrics for processes.

Class (layer)	WMC	DIT	NOC	CBO	RFC	LCOM
`BioData` (abstract)	0	0	3	15	21	0
`AbstractBioData` (common implementation – system)	9	1	4	0	6	0
`BioFile` (abstract)	0	1	1	1	2	0
`AbstractBioFile` (common implementation – module)	24	2	2	2	21	42
`AbstractVTKImageFile` (common implementation – third-party specific)	12	3	5	2	17	25
`Abstract2DImageFile` (common implementation – file type specific)	6	4	4	1	10	15
`PNGFile` (module impl.)	1	5	0	0	2	0
`Abstract3DImageFile` (common implementation – file type specific)	4	4	2	0	5	0
`LSMFile` (module impl.)	13	5	0	0	10	1

Table 13. Results of Chidamber and Kemerer metrics for file data.

With the exception of one class – the colouring process –the results of the measurements are satisfactory. There is an increase in WMC, RFC and LCOM metrics in the common implementation layers caused by the raising complexity, as new functionality and common behaviour is introduced. This increase, however, does not cause the numbers to reach alarming levels.

The value of CBO is high in the abstract layer, where the interactions between components are defined. Furthermore, the coupling for the concrete processes layer is high because of the functionality provided by third-party library.

The layers were created as described in previous sections – by abstracting functionality common to features introduced one after another. This fact is reflected in the DIT and NOC metrics. The former naturally increases as the classes belong to more concrete layers. The classes contained in the concrete layers are not inherited, thus NOC is zero. This metric increases for the intermediate common implementation layers and keeps moderate values for the fully abstract ones.

The class `ColouringProcess` stands out of the abovementioned analysis. The values of RFC, CBO and WMC for this class are unexpectedly high, especially when compared with other classes. The class is large (720 lines of code) and expanded (total of 36 fields), which negatively affects both RFC and WMC. Since processes – in general – are more oriented towards graphical user interface, high CBO can be explained by the complexity of such interface and a high number of graphical components displayed on screen.

The other two metrics, however, indicate that the programmers responsible for implementation of the class produced non-optimised, lengthy code. Similar results were noticed for other classes that deliver complex functionality. Previously we mentioned that the programmers involved in the project were students, who did not have prior occasion to work with construction of large-scale software. This fact justifies poor metric results for classes that implement complex problems.

Regular code reviews were held in order to reduce the number of defects and improve coding style. The classes, for which such supervision was not possible, were of noticeably lower quality. Furthermore, the quality of the code was addressed during daily meetings, in which the conclusions from code reviews were presented to the programmers. We have found that such reviews also motivate the programmers and improve their learning.

16.2. Relation to generally accepted standards

The representative excerpts from the results of the measurement program carried out during the development of *BioImageXD2*, due to the lack of data, cannot be compared to neither previous version of the software nor other systems constructed with SFI. However, we can relate the results to typical values found in other software projects.

During and after the development of *BioImageXD2* we used STAN toolkit [125] for automated metrics collection. For each metric it defines a (configurable) range, which the measured value should satisfy. The default ratings for the metrics introduced by Chidamber and Kemerer are listed in Table 14. For two of the metrics, NOC and LCOM, the defaults are not available, as they vary from one project to another.

Bound	WMC	DIT	NOC	CBO	RFC	LCOM
Lower	0	0	n/a	0	0	n/a
Upper	100	6	n/a	25	100	n/a

Table 14. Default thresholds for Chidamber and Kemerer metrics.

The toolkit enables also gathering the average values for each metric on a system level, i.e. taking into account its every class. The results are presented in Table 15. The full report is available in Appendix 4.

Metric	Value	Number of violations
Number of Top Level Classes	333	
Number of Packages	34	
Average Number of Methods per Class	6.21	
Estimated Lines of Code	27601	
Average Cyclomatic Complexity	1.40 (max 22)	8
Average Distance	-0.19 (min -1)	8
Average WMC	8.70 (max 113)	2
Average CBO	3.84 (max 46)	6
Average RFC	11.01 (max 151)	3
Average LCOM	30.94	

Table 15. System-wide metrics for *BioImageXD2*.

Given the size of the system, the number of metric violations is surprisingly low. Their analysis revealed that the violations are caused only by a few classes that tackle complex problems – which is a result of inexperience of the programmers, as we mentioned previously. The problematic code, however, does not affect the overall quality of the design indicated by the metrics. Most notably, the effects of the violations are not propagated to other elements of the system.

16.3. Code pollution

In order to provide more meaning to the analysis of the results of the measurement plan, we utilise the concept of *pollution* provided by the STAN toolkit [126]. The Pollution of an artefact (package, class, etc.) is a non-negative number that serves as a quick indicator for the amount of metric violations caused by the artefact and its descendants. The pollution gives an impression of a structural quality; however, it depends on the metrics and their preferences. Since we use the pollution as an indicator, we rely on its default settings (standards), as provided in the tool and described in the previous section.

The pollution diagram is drawn as a chart diagram with sections representing metrics that violate the standards. The size of each section corresponds to the pollution given metric causes to the artefact. The pollution diagram for *BioImageXD2* source code is shown in Figure 62.

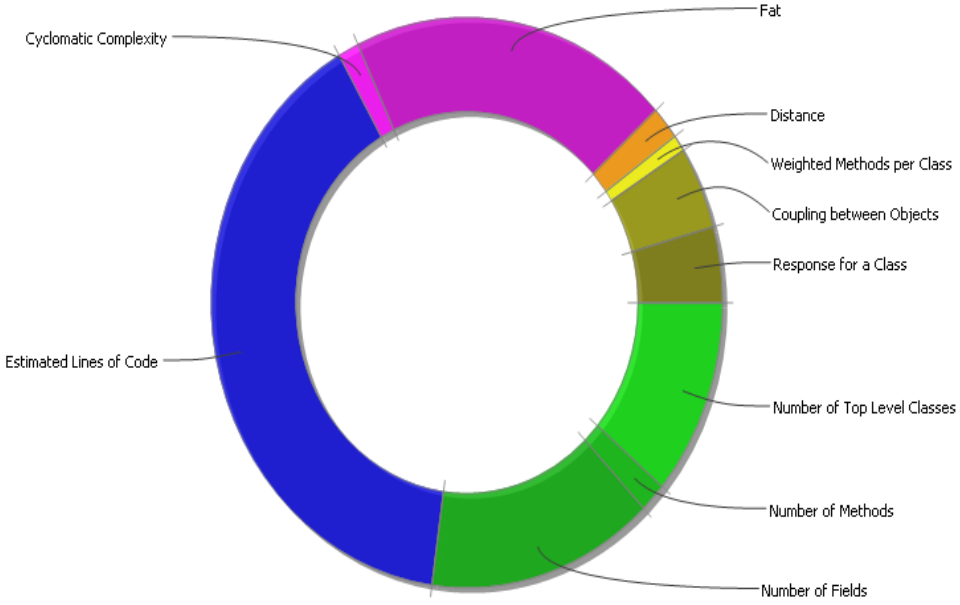


Figure 62. Pollution diagram for *BioImageXD2*.

As it can be noticed, the metric Estimated Lines of Code is violated the most and contributes the most to the pollution (39%). This metric is applied to classes and methods. It approximates the number of source code lines of an artefact, excluding comments, empty lines and import statements. A method violates this metric if it contains over 120 estimated lines of code; for classes the limit is raised to 400.

Other violations are less significant in terms of pollution: Fat metric contributes to 20% and Number of Fields to 14%. The latter is applicable only to classes, whereas the former also to packages. The Fat metric of a package is the edge count of its unit dependency graph, which contains all the top-level classes of the package. For a class this metric is the edge count of its member graph, which contains all fields, methods and member classes. The Number of Fields metric is simply the number of attributes declared in a class.

The results of all three metrics straightforwardly indicate that the code is too expanded, i.e. it contains too many lines of code. The fact that the programmers responsible for the code were not professionals, but rather non-experienced students, must be taken into account. The complexity of the code produced by the programmers was excessively high, especially whenever the solution to the problem was difficult or required analysis of a number of different scenarios. These issues, however, resolved with time, as the Team gained experience and were instructed to focus more on optimising their solutions.

16.4. Perception of the developers

Subjective perception of the developers was gathered in addition to the measurements of the source code for the project *BioImageXD2*. An online survey, presented in Appendix 1, was carried out after the development finished.

The three members of the development team and the Product Owner were asked to fill a questionnaire of 27 questions divided into 6 groups. The questions were detailed; therefore the survey can be seen as a replacement of face-to-face interviews, especially given such a low number of participants. We focus on major findings from the survey in this chapter; more detailed analysis is presented in Appendix 2.

The Team

A significant part of the survey focused on the suitability and the evaluation of the development process used with *BioImageXD2*. The process was in general well perceived by the Team. The developers underlined the possibility of adjusting the process to their liking, as well as pointed high potential for reuse in future projects of similar complexity.

Minor problems were reported in understanding the process by one of the Team members not familiar with agile development philosophy. However, once the developer was given more information about the process and the paradigm of SFI, no further concerns were reported.

The organisation of work on the project and communication between the programming team and the customer was well supported by the development process. Its agile nature further helped in overcoming the difficulties of the development and adjusting it to the changes in requirements and setting.

Product Owner

The survey indicated certain problems related to the role of the Product Owner. The reported troubles concerned mostly the adaptation to the development process and were further supported by the Product Owner rating the process and its suitability low. Based on the results of the survey and our experience, we can indicate a possible origin of the situation.

We assume that the problems may have been caused by the lack of detailed information about the progress of the development, despite the fact that such data was available online through the issue tracking system. A solution to this problem might be to provide more explicit interaction and information exchange between the Team and the Product Owner. This can be achieved e.g. by shortening sprints. However, in applications as complex as *BioImageXD2* short development cycles may not be possible. An introduction of short mid-sprint progress meeting with active participation of the Product Owner might be seen as an alternative. This would give the Owner more control over the development and also allow the Team to deliver more usable software.

16.5. Evaluation of the measurements

The measurements we performed and presented previously clearly show that the architecture of *BioImageXD2* supports code reuse. The layered design and low complexity of code introduced in each layer enables the

code to be learned and understood without additional effort, which in turn leads to increased maintainability. The components with clearly defined responsibilities and a limited number of dependencies enable the system to be easily extended, thus contributing to the overall adaptability of the constructed software.

BioImageXD2 is a system that is based on its predecessor with respect to commonly used features, but with redesigned and refactored architecture. No quality assessment was performed during the development of the previous version of the software, hence we may not assess the impact of SFI on the quality of the software. Our most valuable data is the subjective perception of the developers involved in the construction of both systems, which we presented earlier.

Based on the collected evidence we presented, it is clear that the goals set for the reengineering of *BioImageXD* were reached. The application of Stepwise Feature Introduction resulted in constructing a system that is of good quality. The software we built according to the paradigm is maintainable, reusable and adaptable. Moreover, according to the measurements, the architecture conforms to the best practices of object-oriented design and is up to the object-oriented standards.

The strongest statement we can make in these conditions is that applying SFI to the development allows creating a system that adheres to the principles of object-oriented design and is reusable and maintainable. Moreover, it does not seem to disrupt the rules of design and negatively affect complexity of produced software. As the paradigm was not evaluated previously at all, we consider these results to be a significant achievement.

17. SFI and object-oriented design

Object-oriented design is the process of planning a system of interacting objects for the purpose of solving a software problem [188]. It is a natural choice when object-oriented programming languages are to be used for the implementation. The paradigm of SFI was designed to use such languages [8], hence it is essential to ensure that the application of the paradigm does not violate the basic principles of design. The analysis presented in this section concerns the extended paradigm; the results, however, are also applicable to the unmodified version of SFI.

The basic constructs of object-oriented programs, objects, are utilised by SFI to introduce features to the system in construction. Each object (or its class) encapsulates one, and only one, feature. Inheritance is an essential characteristic of object-orientation. It corresponds to feature extension and, for languages that support multiple inheritance, also feature combination. This mechanism provides a natural and straightforward opportunity for working with different features of the system.

Stepwise Feature Introduction is a paradigm that provides an organised way of constructing software. Object-oriented programming, on the other hand, is a method of representing a software system as a collection of interacting objects. Therefore, SFI is an addition, not a replacement, to object-orientation.

In object-orientation packages provide a way to organise software in terms of general functionality. The principles of object-oriented design are of use when grouping related classes together to promote code reuse and reduce the complexity of a system. The packages play no other role in object-oriented programming. The purpose of layers of SFI is similar, but a different method is used to achieve it. Moreover, layers are an essential structure of the paradigm.

Stepwise Feature Introduction groups classes together based on their relation to a service provided in each layer. This grouping is not caused nor affected by the design guidelines. Adhering to SFI ensures that each layer delivers well-defined and self-contained functionality that can be further extended. Our objective here is to show that layers created in such manner adhere to the principles of object-oriented design, which are vital for the construction of high quality software systems.

The principles of object-oriented design are intended to make the resulting software reusable, manageable and robust. These design principles formed as a result of work of many researchers and engineers, including B. Meyer, B. Liskov and R. C. Martin. They can be seen as indicators of *good style* in design of object-oriented software systems. While adhering to these principles is not a necessity, it often provides significant benefits to the system, increasing its reusability or robustness. The design of both systems presented in the thesis follows these rules.

There are five main principles concerning the design of classes, three regarding the design of packages and three about coupling between packages [109], as shown in Figure 63. Other design guidelines have also

been proposed [114][83][89], but we consider them to be the consequences or variants of the eleven design principles we focus on in this section.

Figure 63. The principles of object-oriented design.

17.1. Class design

The requirements of a system often change during its development and also after it is released and used. The *Single Responsibility Principle* states that because each functionality is an axis of change, it should be a separate class. A class should have one, and only one, reason to change, so that when a modification to software requirements causes this class to change, the remaining parts of the system are not affected. Furthermore, a modification to database schema, graphical user interface, report format, or any other segment of the system should not force that class to change [98]. In other words, each class should have a clear and consistent functionality it is responsible for.

Adhering to the Single Responsibility Principle when designing classes ensures that each class has one, precise functionality. SFI follows this principle naturally, as each feature introduced to the system encapsulates a small, well-defined piece of functionality in a class. Therefore each class has a single responsibility and only one reason to change.

The *Open-Closed Principle* requires that software entities are open for extension, but closed for modification [114][99]. This means that all new functionality can be achieved by adding new subclasses or methods, or by reusing existing code through delegation [91]. This principle, in fact, should be satisfied by all the software systems, not only the object-oriented ones, by replacing inheritance with similarly behaving mechanisms.

The paradigm of Stepwise Feature Introduction can be seen as a more general version of the Open-Closed Principle. New features are added

to the existing system by extending existing services (in object-oriented terminology: inheritance), adding new service users or combining existing features (multiple inheritance or delegation). SFI does not prevent the developers from changing the source code – and neither does the principle – instead the paradigm provides an organised way of extending the existing code.

The *Liskov Substitution Principle* defines substitution of subtypes by stating that an instance of derived class should be able to replace any instance of its superclass, without the user of the class noticing any difference [90][92]. This principle allows subclasses to be used in the context of their superclasses. Moreover, the tests of the superclass can be executed for its subclasses as well [100]. This principle can be seen as the fundamental principle of object-orientation.

The Liskov Substitution Principle defines the substitution of types within one hierarchy. The paradigm of SFI requires that the old functionality is preserved whenever new features are added or existing ones are combined. This in turn guarantees that the classes containing new functions can be safely used whenever their parent classes are needed, with the same result. It is important to emphasise that the paradigm itself does not enforce any method to do the actual check. It is up to the designers and developers to provide such tools, be it regression testing or formal verification.

The *Interface Segregation Principle* states that the dependency of one class on another should be restricted to the smallest possible interface [101]. A common understanding of this principle is that entities should depend on as little, as possible. The reduction of information that one class needs to know about another is a consequence of the Single Responsibility Principle, adhering to which causes a class to be specialised and have a well-defined functionality. The Interface Segregation Principle concerns only the dependencies between classes – instead of relying on the whole exposed interface a class should depend only on what it needs to perform its functions.

As we said previously, in Stepwise Feature Introduction a class is a service provider or a user or both. The interface to the class is determined by the service it provides. As the features are introduced one by one, in small yet fully executable increments, the interface is thus as small as possible in the current circumstances. Such construction of the system

allows service users to depend on a small, well-defined interface of the provider, and therefore conforms to the Interface Segregation Principle.

The *Dependency Inversion Principle* forces the implementation details to depend on abstractions [93]. The entities that implement a high-level policy should not depend on the modules that implement the low-level ones, but rather they both should depend on some well-defined interfaces [97]. This principle is most commonly applied when designing interfaces to classes. Such interface is defined based on the desired functionality of a class, rather than on the implementation details. In other words, the interface of an entity is what other entities should rely and depend upon.

SFI naturally provides a mechanism to adhere to the Dependency Inversion Principle. Each layer encapsulates a feature, which is made available by one (or more) service providers. The users of such service can only depend on its interface, not on the implementation. Moreover, with respect to the overall system functionality, layers introduced earlier are more general and contain less functionality than the later ones. This helps to retain the correct direction of dependencies even more.

17.2. Package design

The *Reuse-Release Equivalence Principle* deals with the design of packages, which group related classes together. It states that the granule of reuse is the granule of release [94]. In order to effectively reuse code in a different software project, this code must arrive in a complete black box package which is to be used, but not changed [102]. The rule also claims that the granule of reuse-release is the software package.

The software built according to SFI organises the software in layers, which can be used by other layers or classes, only based on the available interfaces. The layer itself cannot be changed. We can thus equate the terms package (with respect to the design principles) and layer. In other words, in terms of SFI, the granule of reuse and release is the layer; hence the Reuse-Release Equivalency Principle is fulfilled.

The *Common Closure Principle* says that classes within a released component should share dependencies and be related to each other. That is, if one of them needs to be changed, they all are likely to need to be changed [94][103]. This principle prevents tightly coupled classes to be released in different packages. Moreover, adhering to the Common

Closure Principle minimises the propagation of a change internal to a package to different parts of the software.

The tightest dependency between classes developed with SFI is between a feature and its direct users. As mentioned earlier, such classes together belong to a single layer and are released together. Therefore, the Common Closure Principle is preserved.

The *Common Reuse Principle* is a consequence of the reuse-release equivalence [104]. The classes in a package are reused together; if one of the classes in a package is reused, all of them are [94]. More generally, the dependencies of the classes of the reused package are inherited by the software part reusing it.

Although in SFI it is possible to directly reuse a service provider, due to not knowing the dependencies between such provider and other classes in its layer, a whole layer must be depended upon. In other words, relying on a feature contained in a layer results in the propagation of such dependency to all the classes contained in the layer. Therefore, a class using a service provider in fact depends on the whole layer – which follows the Common Reuse Principle.

17.3. Package coupling

The *Acyclic Dependencies Principle* is onq of the three principles that concern package coupling. It states that there must not be any cycle in the dependency structure of the packages [94] [105]. More precisely, the directed graph with nodes corresponding to packages and edges to the dependencies must be acyclic.

The layers of software built according to the paradigm of SFI may only depend on the functionality available at the moment of introducing them to the system. As features are introduced one after another, rather than in parallel, there is no risk of creating a cycle in the dependencies between layers, thus satisfying the Acyclic Dependencies Principle.

The *Stable Dependencies Principle* enforces the dependencies between packages to be in the direction of the stability. A package should only depend upon packages more stable than it is [95]. *Stable* means here *hard to change* [106].

When developing software system incrementally with SFI the simplest and the most crucial features are usually implemented first. Customer feedback after every iteration of the development process

ensures that these core features rarely change once they have been extended. On the other hand the features that have been added recently are more likely to be revised after obtaining the customer feedback. The layers with such features are easy to change as such modifications do not affect other parts of the system. We consider layers containing such features as instable, as opposed to the stable ones containing the core functions. Therefore, the direction of the dependencies follows the Stable Dependencies Principle.

The *Stable Abstractions Principle* states that the abstractness of a package should be in proportion to its stability [95]. For a package abstractness is calculated as a ratio of the count of the abstract classes it contains to the number of all the classes. Therefore, a package that contains only abstract entities should have maximum stability i.e. should be nearly impossible to modify. Adhering to this principle yields similar results as the Dependency Inversion Principle for class design – the more concrete packages depend on the abstract and stable ones. As a result the changes to the instable packages do not propagate to the abstract packages.

The Stable Abstractions Principle is a consequence of the Stable Dependencies Principle and is related to the Open-Closed Principle [107] and, as such, it is also preserved by SFI. We have previously stated that the earlier layers contain less functionality, while the recent layers are the opposite – they extend previous features and provide more detailed behaviour. As we look through the hierarchy of layers, the later the layer was added, the more functionality it contains or utilises. The stability of the layers behaves in the opposite way; hence we can state that the structure adheres to the principle of stable abstractions.

Part VI:
Discussion

18. Related approaches to software construction

The goal of designing a system is, in its most general sense, to identify parts of software and define communication between them. Object-oriented design helps in decomposing system into objects, which later interact by message passing. However, the principles of design recognise that certain issues are related to a number of different objects and do not clearly fit such an approach.

The design patterns [61] can be used to solve or simplify common design problems, related in particular to structuring and modelling the system behaviour. Interestingly, these problems often arise due to the features of object-orientation itself [65]. The majority of them is not found when using a different programming language [120] or an alternative design approach. Furthermore, certain object-oriented design patterns, like Model-View-Controller [143], are in fact architectural styles and predate the concept of pattern by several years. Therefore, they can be applied regardless of the used design approach.

Stepwise Feature Introduction can be seen as a development philosophy that accommodates the evolution of a software system. It not only states the principles on how software should be structured, but also defines methods of extending its functionality. We have shown that the object-oriented programming languages are a natural choice for the paradigm. In this chapter we present other approaches to system development that can be seen as alternatives to SFI. Moreover, the methodologies we describe can usually be represented in terms of objects and implemented in object-oriented programming languages and thus incorporated within the paradigm.

18.1. Aspect-Oriented Development

The objects in object-oriented design represent the primary functionality of a system. However, certain requirements cannot be clearly separated in this manner and span over a number of objects or packages. Aspect-Oriented Development [78] is an emerging software development paradigm, which seeks to establish new modularisation schemes. The primary focus is on *aspects*, which represent distinct concerns précised in the requirements, and *join points* in the code that combine the behaviour provided by aspects.

Aspect-orientation focuses on the identification, specification and representation of cross-cutting concerns, i.e. those requirements that span across multiple parts of the domain model.

Components and aspects

The major modularisation principle of Aspect-Oriented Development is based on whether or not a given property of a system can be cleanly implemented. Clean implementation means that the code is well-localised and easily accessed. Components are typically units of functional decomposition of the system [78] and – by definition – can be cleanly implemented. Moreover, the implementation may be done in any programming language, in particular an object-oriented one.

Not all of the properties of the system can be represented this way. This is caused by the fact that most programming languages and development methodologies offer only one mechanism for decomposition of a system into subsystems. Those parts of the functionality that cannot be cleanly implemented are called aspects of the system [78].

Aspects are thus perceived as properties of the system that influence – or cross-cut – a number of components. Such perspective can be used to explain e.g. the existence of different extensions to pure object-oriented languages, like dynamic scoping or exception catching mechanisms [78]. These extensions help programmers implement certain aspects of the final system. Aspects cross-cut components; hence their implementation may require a dedicated programming language.

Join points and aspect weaving

Aspects and components of a system can be implemented using different programming languages. The resulting executable program must combine the behaviour of aspects with the one of components. Therefore, the languages used for components must have syntax that allows aspects to coordinate with them. Such elements are known as join points. These points do not have to be explicit constructs; rather they are clear, perhaps implicit, elements of component semantics [78].

The process of generating a join point representation of components and execution (or compilation) of aspects with respect to it, is known as aspect weaving. The aspect-oriented programming languages can be designed to allow weaving either during compilation or at run-time.

Aspects and objects

Aspect-oriented development can be seen as a domain-specific language built on top of the programming language [58][170]. Furthermore, it is a technique complementary to object-orientation [175] and can be represented in object-oriented terms. The components of a system can be implemented using objects and mechanisms common to object-oriented programming, like inheritance and composition. The aspects, on the other hand, should be seen as collections of methods that extend the functionality of a component. Such approach to design promotes concept separation, code reuse and increases the overall maintainability and understandability of the final system.

Aspect-Oriented Development versus SFI

Aspect-oriented development proposes a modularisation of concepts that is different from the one used in object-oriented design and thus in SFI. Certain cross-cutting concerns – logging, security auditing, transactions, multithreading or graphical user interface – are present in the majority of computer applications [157]. By targeting specifically the representation of such concerns aspect-oriented development is beneficial to the construction of software systems.

The focus of aspect-oriented development is on the decomposition of requirements and their interactions. It promotes code reuse, enables better encapsulation and increases maintainability of the system. The evolution of the software system, once it is built, is not covered. However, aspect-oriented development aims to isolate the non-changing domain knowledge from frequently modified requirements, which facilitates adding and changing functionality after the system is constructed.

Aspect-oriented development imposes an additional restriction on the language, i.e. the ability to describe and weave aspects. The support for aspect-oriented development has been added to the majority of the modern programming languages. In some cases, however, there are significant changes over the syntax of the original language. This may impact the development process negatively, as the developers need to learn and adapt to the new setting. Stepwise Feature Introduction, on the other hand, places no additional constraints on any object-oriented programming language and thus does not require the additional effort to construct, adapt or modify an existing language.

Aspect-Oriented Development and SFI

The paradigm of Stepwise Feature Introduction does not enforce a particular development methodology, yet it suits best object-oriented design and programming. Therefore, it is suitable for designing and implementing components in aspect-oriented development, where traditional development methods can be used.

The aspects and the process of weaving, on the other hand, require a different approach. The elements of a system built with SFI may be service providers, service users or both. However, with aspect-oriented development the components do not use the service provided by the aspects. In fact, functionality of an aspect is merged with the one of a component. Therefore, at run-time there is no distinction between the service provider and service user.

In order to be efficiently used with aspect-oriented development, the paradigm of SFI should be modified. More precisely, the relation between service providers and service users must be reworked. However, on a general level aspect-oriented development is possible with the paradigm as it is described in this thesis. Each layer of the system can contain components and aspects, which are then combined by the weaver.

18.2. Data, Context and Interaction

A runtime structure of an object-oriented program often bears little resemblance to its code structure [61]. Furthermore, it is difficult to reason about the behaviour of an object-oriented system based on its code [61]. This dissonance between the static code structure and its dynamic representation at run-time negatively affects the quality and usability of the software. Data, Context and Interaction is a paradigm for development of object-oriented systems [142], aimed at reducing this gap. The central concepts of the paradigm are, as implied by the name, data, context, and interaction.

The original intention of Data, Context and Interaction was to allow more efficient modelling of the model part in Model-View-Controller [143] architecture. However, it is more general, as its principal idea is to separate the static code that describes state of the system from the dynamic code that is responsible for its behaviour. This goal is achieved by organising code into different perspectives, each focused on certain properties of the final system [142].

Data

The Data perspective represents the static part of the system, i.e. its domain model. It is implemented with classes that should contain only the primitive operations on the data. In other words, the classes should not have any functionality that corresponds to any particular use case.

Context

A property of the final system is, at run-time, executed by a network of interacting objects. Contexts are responsible for constructing such network, in terms of roles different objects play in it [142]. Each Context encapsulates a use case, or a part of it.

The Context is the class (or its instance) that includes in its code the roles for a use case it implements. Moreover, it must also contain the code that maps these roles into objects at run-time to be able to execute the use case [144][189]. In other words, Context must be able to locate or construct objects that will be put to their roles in a scenario it represents [144], as shown in Figure 64.

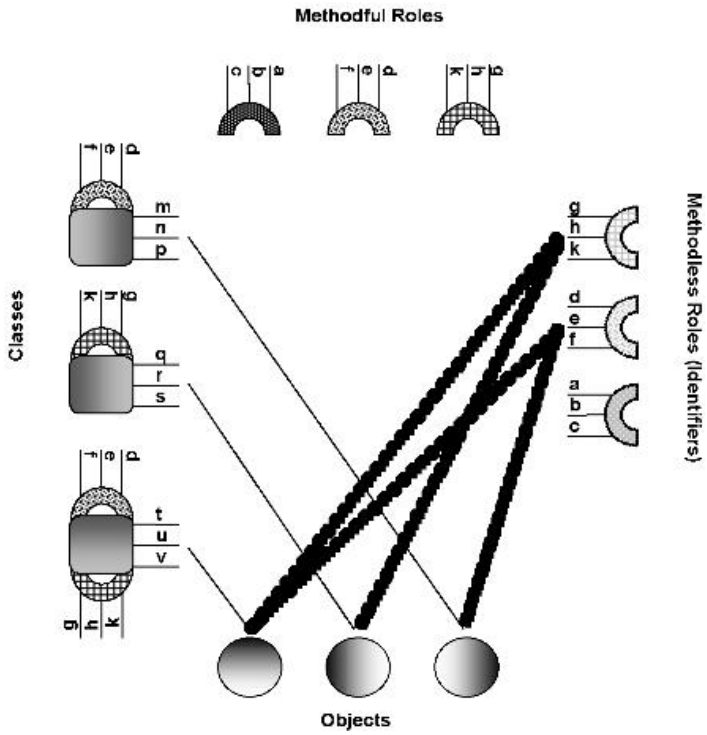


Figure 64. Combining roles and objects in Data, Context and Interaction.

Interaction

The Interaction perspective focuses solely on the behaviour of the system. It describes end-user algorithms in terms of roles different objects play, not in terms of the objects themselves [144]. The code specifies how roles collaborate with each other to realise a property of a system [142]. This is achieved by specifying stateless roles [189], or through role methods, that are injected into every object that realises a given role [142]. In other words, the dynamic code of a role is added to the static code of data, based on a current context. Within a given Context each role is always bound to a single object; however, it is possible that one object plays many roles at the same time [189].

Data, Context and Interaction versus SFI

The paradigm of Data, Context and Interaction, in many respects, unifies a number of approaches that appeared around object-oriented development and programming [144]. Moreover, it seamlessly fits agile development [16], as it separates the stable domain knowledge (Data) from the rapidly changing use cases (Context and Interaction) [144]. Such combination also supports future extensions and modifications of a system once it is built.

Due to the fact that Data, Context and Interaction injects the behaviour to run-time objects, it is suitable for modern dynamic, object-oriented programming languages, like Ruby [57]. More static languages, including Java [132], often do not support or support partially run-time modification of existing objects. Hence, the application of Data, Context and Interaction to the development using such language may not be possible. Stepwise Feature Introduction, on the other hand, relies solely on subtype polymorphism and inheritance and may be thus applied to a vast majority, if not all, of object-oriented programming languages.

Data, Context and Interaction focuses on a construction and evolution of a single software system. The paradigm of SFI produces a collection of reusable and executable software systems. Therefore, it appears more appropriate to use SFI whenever product lines are of concern.

Data, Context and Interaction and SFI

We have said previously that in SFI a class is a service provider, service user, or both. The domain-knowledge part (Data) is static and rarely changes. The Data classes do not benefit from any particular service;

hence they are not service users. They do not carry any functionality either, beside the simplest and the most primitive. Therefore, the only service they may provide is the access to the information and the ability to modify it.

The Interactions specify the behaviour of the system and different roles objects may play. In SFI terms they are service providers. The Contexts, on the other hand, are service users only, as they benefit from both Data and Interactions.

A system built with SFI has a layered architecture, with each layer providing a well-defined increment in functionality. We can map such increment to a Context, which represents a particular use case. With increasing level of abstraction Contexts can be used as domain objects (Data) [144], thus giving rise to a layered hierarchy of Contexts, each providing more functionality. Contrary to SFI, however, such hierarchy would be based on usage, rather than inheritance.

Finally, Contexts are capable of binding roles to different objects and triggering the interaction between them. Therefore, they ensure that the system is executable after each increment and thus establish the essential property of a system built with SFI.

18.3. White- and black-box frameworks

Software frameworks are of key importance when developing large-scale object-oriented systems. They offer higher productivity and promote reusing both the design and the code [145].

The use of frameworks with software of significant scale is often beneficial, as they abstract certain processes and patterns that are common to a wide range of systems [174]. This often means that the framework controls the execution of the software. Using the framework results in not implementing parts (or all) of the intended behaviour of the software, instead relying on the framework to provide it.

Based on a way a software system utilises framework-provided functionality, we can identify *black-box* and *white-box* frameworks. As it can be expected, very seldom a framework can be categorised solely as black-box or white-box. Typically, real-life frameworks combine these approaches; thus, a framework often shares properties of both [145].

Black-box frameworks

In black-box frameworks the classes provided by the framework can be readily instantiated, meaning that they contain concrete code. As a result,

the framework as a whole can be used as-is. The use of a black-box framework is thus mainly done through composition of objects provided by it [77][145]. While presenting *BioImageXD2* we mentioned VTK [81] and ITK [80] image-processing libraries. They have all the properties of black-box frameworks and can be perceived as such.

Frameworks of this kind do not require any knowledge about their internal structure, apart from precise description of classes that are to be instantiated in the custom code. Typically, this is achieved by extensive documentation of use cases for each such class. Moreover, black-box frameworks usually do not affect the design of software that uses them.

White-box frameworks

White-box frameworks, on the other hand, represent general model of a certain domain. The custom code is required to inherit parts of that model in order for the framework functionality to be utilised [77][145]. In languages that do not support multiple inheritance this significantly influences the design of the system. The software that uses the framework is often designed to be an extension, or a part, of the framework it uses.

Such approach requires the users to know not only the domain model, but also the internal structure of the framework. As opposed to black-box frameworks, white-box frameworks are often shipped together with the source code in addition to documentation.

BioImageXD2 is an example of a white-box framework. It models image processing and analysis and allows introducing new behaviour by creating subclasses of existing components.

Frameworks versus SFI

The major advantage of using frameworks is their abstraction of certain repeatable patterns that occur in many large-scale software systems. A typical examples include logging the execution trace, communicating with external databases, presenting the user interface, and so on.

The frameworks aim at simplifying the software that is being constructed. This is evident especially in the case of black-box frameworks, which provide ready-made solutions and usually do not affect the design of software that uses them.

SFI, in principle, operates on a higher level of abstraction. It deals with software development in general. The paradigm is supposed to simplify the process of constructing the software, not directly the software

itself. By this we mean that while we expect the system constructed with SFI to be simpler, maintainable and reusable, it will still need to implement all the required functionality. For that reason whenever implementation of a system should be simplified by removing some of its parts or delegating some of its functions, frameworks are a better solution than SFI. They are also more suitable in settings with fixed set of requirements and an established development process, as the paradigm, to some degree, affects those aspects of system development as well.

Frameworks and SFI

Due to high level of abstraction of SFI, frameworks can be utilised within the paradigm. More importantly though, the software produced with SFI can be seen as a framework in itself. Any (sub)system created with the paradigm represents a simplified domain model (compared to subsequent development iterations). Its functionality can be customised and extended by extending existing classes (white-box framework) or by combining them (black-box framework).

Defining framework as an external entity that abstracts certain repeating patterns and represents a (customisable) domain model means that it is possible to apply the definition to other areas as well. For example, Scrum [154] is a process framework, within which different techniques can be applied to construct software.

In this sense, SFI is also a framework for agile software development that covers all aspects of the development. The principles of the paradigm outline the resulting architecture of the system (based on layers), propose a process framework (Scrum) and means to ensure the quality of the constructed software (correctness and testing). SFI has properties of both white- and black-box frameworks. While certain aspects of the paradigm, like development process or quality assurance, can be modified by providing alternatives, SFI can be also used as-is, without any changes.

18.4. Feature-Driven Development

Feature-Driven Development is an iterative and incremental agile software development process. It incorporates a number of industry-recognised best practices to deliver tangible, working software repeatedly in a timely manner [193][42].

Best practices

The practices that Feature-Driven Development builds on are all driven from a client-valued feature perspective. There are eight practices present in the core of this development method [42].

Domain Object Modelling provides a framework which is later extended by adding features. It is constructed by exploring and explaining the domain the software will be applied to.

Developing by Feature means that any requirement that is complex and is estimated to take more than two weeks to implement should be divided into smaller parts, each called a feature. Focusing on relatively small and manageable pieces of functionality allows the software to be constructed in short iterations.

Individual Code Ownership distributes the code among the developers and makes each developer responsible for owned parts of code. This does not prevent other people involved in the development from modifying code. Rather, it makes a selected developer responsible for performance, consistency and integrity of the code he or she owns.

Feature Teams are dynamically formed to develop a small increment in functionality. The teams are formed and disbanded as the need arrives, however, as a rule, each feature should be implemented by a group. This practice ensures that design decisions and implementation are made collectively and that alternatives are considered.

The primary role of *Inspections* is to ensure high quality of the code. The developers that perform code and design reviews focus on detecting defects and identifying those parts that do not meet quality standards set for the project.

Configuration Management, as the name implies, focuses on managing the implemented features. Moreover, its goal is to identify which parts of code are responsible for a given feature. Finally, the purpose of Configuration Management is to keep track of changes done to each class and to the project as a whole.

Maintaining *Regular Builds* ensures that there is an executable version of the system that is up to date and can be presented to the stakeholders. Regular Builds also help in detecting integration errors and provide constant feedback about the direction of development.

The final practice – *Visibility of Progress and Results* – is common to most agile development processes. Not only it motivates the developers, but also helps managers in steering the project in the right direction.

Feature-Driven Development versus SFI

SFI is a more detailed paradigm than Feature-Driven Development, in terms of the areas of the development it covers. For example, the latter does not place any constraints on the architecture of the developed system or its development process, neither it ensures that the correctness is preserved from one iteration to another. For the above reasons Feature-Driven Development is a more suitable technology for developing non-layered or non-object-oriented systems.

Feature-Driven Development and SFI

As an agile process, Feature-Driven Development shares certain characteristics with SFI. Most notable ones are the division of functionality into small, manageable steps and the requirement of having an up-to-date, executable version of the systems.

Due to the nature of SFI another part of Feature-Driven Development – Configuration Management – is straightforwardly provided. SFI has a precise definition of a feature. Furthermore, the classes in the system play roles of service providers and service users. This enables features to be tracked and thus binds functionality with code that implements it. In addition, the layering of the system allows the changes and modifications to be easily identified.

A closer look at the development of *BioImageXD2* reveals that, in fact, all of the practices of Feature-Driven Development can be incorporated into SFI. This not only proves the versatility of SFI, but also is a strong argument for applying it in practice.

19. Conclusions

In the thesis we focused on presenting and evaluating the paradigm of Stepwise Feature Introduction and its suitability to the development of large-scale software systems. We have illustrated the basic concepts of the paradigm and explain the key characteristics of software built according to SFI with our pilot case study, *ReThink*. Then we have described *BioImageXD2* and its layered, modular architecture and its components to provide the context for the analysis of the quality of the product. We also confronted our findings with the subjective perception of the developers and generally accepted standards in object-oriented software development.

SFI fits well iterative development processes, in particular Scrum. We have shown that different concepts of the paradigm match the ones of the process. Furthermore, we proposed an extension of Scrum in order to have better control over development and integrate it with the paradigm.

Based on the above considerations and the evaluation presented in the thesis we can state that the paradigm of Stepwise Feature Introduction provides the rigour needed for the development of object-oriented systems. The paradigm can be applied to the development on different levels of abstraction. Requirements, components, layers and features enable better and clearer communication with the customer or within the development team. Moreover, the layered structure of the system – or rather, a collection of systems – allows the code to be more easily extended and modified. Finally, the application of agile development methods to the construction gives all the stakeholders more control over the development. Therefore, we can solve our primary research problem and state that Stepwise Feature Introduction is suitable to the development of large-scale software systems.

19.1. Overview of research projects

Our work is practical and strongly related to software construction and engineering techniques, therefore a project-driven approach was chosen as the most beneficial. The SFI framework we presented in this thesis has been successfully applied to the development of two software systems that varied in purpose, scale and environment.

The application of Stepwise Feature Introduction organises the software in layers. Each layer provides a well-defined increment in functionality of the whole system and constitutes a point, from which future modifications may derive. Furthermore, the application of the paradigm results in a collection of different systems, as the software must be executable after each added layer.

The collection of systems is evident in case of *ReThink*, as the game is available for various platforms (mobile phones, web browsers, desktop computers and text terminals). The core functionality is organised into layers shared among the deployment platforms and the server needed for online gaming. The platform-specific code extends this hierarchy at different levels, depending on what functionality is required for each platform.

The architecture of *BioImageXD2*, on the other hand, is more modular. The system contains a number of interacting components

(modules) and enables its users to construct a pipeline for processing digital images. The layering scheme is not as apparent as in the case of *ReThink*, due to significantly greater complexity and distinct architecture, yet it is still an important part of the design. The dedicated system executable is responsible for constructing the graphical user interface and setting up the environment, in which the modules operate. Furthermore, it can be configured to allow the execution of the system at a certain layer or to include a limited set of modules.

19.2. Extensions of SFI

Prior to the research presented in the thesis the paradigm of Stepwise Feature Introduction had been applied in practice to the development of medium-sized software systems, in a controlled environment [116]. The construction of large-scale software requires the paradigm to be adjusted according to the project requirements, e.g. in the area of daily routine or overall system design. The general development approach proposed by the paradigm, however, does not change, regardless of the project domain and complexity.

The requirements of *ReThink* forced minor modifications to the SFI framework, in addition to what we presented in the thesis. High specialisation of team members was caused by the variety of platforms, for which the game was to be deployed. Furthermore, the overlapping requirements of *ReThink* required careful iteration planning. The order, in which the features were introduced to the system, was thus decided before the first iteration.

The complexity of *BioImageXD2* resulted in a number of significant modifications of the paradigm. The high-level architecture of the system – its components and relations between them – was designed upfront through prototyping, which may be seen as a contradiction of agile development philosophy. However, the architecture was designed to be modular, changeable and extendable and thus suitable for SFI.

The existence of the architecture and the interface layer resulted in an extension of the paradigm of SFI. In traditional SFI development a class is a service provider, service user or both. However, the interfaces used to declare components of the system neither provide, nor utilise, any real service. Instead, they precise the ways in which the service ought to be used and implemented. Furthermore, the interfaces define the structure of code, which must be followed in the implementation; hence the name service

borders. An important observation should be made at this point. While the interfaces themselves are not executable, on the abstract level it is still possible to refer to them as providers or users of services their implementations provide.

Software execution as part of testing was another modification to the development paradigm, as it was impossible to effectively and efficiently test certain features of the developed systems. The inclusion of this additional step in the development cycle proven to be successful and straightforward, thus confirming the flexibility of the paradigm.

19.3. Threats to validity

We are aware of several limitations and shortcomings our research had. The most important consideration is that we have presented only two case studies. Although they were sufficiently different to allow generalisation, at least to a certain degree, it is not possible to thoroughly evaluate SFI without developing more software with it.

Despite being developed for external stakeholders, both systems were constructed in a controlled academic setting, with students doing the majority of programming. Therefore, the conditions in which the software was developed differed from the industry standards.

There is not enough data on other projects with similar scale developed with SFI to compare our results with. The systems, to which the paradigm was applied previously, were relatively small and served more as a proof of concept rather than a real-life software systems [9].

After the development of one of the projects we present in this thesis finished, we conveyed a survey among the involved people. We used its results to indicate areas of the paradigm that require a closer look. However, due to the limited number of participants the survey bears no statistical significance and general conclusions cannot be drawn.

Finally, the observations and conclusions we make in the thesis concern the modified version of the paradigm, not the original one. As a consequence, it is impossible to assess our improvements compared to the original version of SFI. Instead, we aimed at showing that SFI in its modified state suits the development of large and complex software systems and enables construction of reusable, maintainable systems.

Due to our work based on only two projects, we anticipate that the paradigm may require even more changes to fit other specific settings and

requirements. Nonetheless we consider our improvements to be a solid step forward in applying SFI to construction of complex software systems.

19.4. Future work

The results of our research open new possibilities for further work. These are determined by the type of the projects our technique was applied to. At this point we can indicate two main areas, which should be investigated, namely architectures and development processes.

Since Stepwise Feature Introduction was successfully applied to the projects presented in the thesis, we would like to evaluate the suitability of the paradigm to development of other kinds of systems. In particular, we are interested in construction of modern, web-based database applications. This type of software gains increasing popularity due to the phenomenon of Web 2.0 [156][64].

The requirements set for Web 2.0 systems often change during the development or shortly after the deployment; moreover, such software needs to be reliable, scalable and extendible. Based on our experience we believe that the paradigm would offer stability and contribute to the reusable architecture; however, we would like to collect evidence to support such claim and examine the degree of improvements. Furthermore, a number of Web 2.0 applications experience a rapid grow in the number of users, development personnel and required resources. Thus, it would be possible to evaluate the paradigm when the development and run-time environments are rapidly scaled up.

Web 2.0 applications are often implemented with dynamically typed languages. The paradigm has been successfully applied to a development with such language [9]; therefore we can conclude that the choice of programming language is irrelevant, as long as object-orientation is supported.

The databases of Web 2.0 applications are often modelled purely in object-oriented terms, with the help of object-relational mappings. However, on the enterprise level there is often a need to model a database with a dedicated language. The applicability and suitability of Stepwise Feature Introduction to legacy database systems needs to be examined. We expect the results to provide an insight on amendments necessary for the paradigm to support development of non-object-oriented systems.

One of the contributions of the thesis is the agile process that matches the paradigm of Stepwise Feature Introduction. However,

we anticipate that in certain cases this approach may not be suitable. For instance, development of *BioImageXD2* required prototyping in addition to regular agile development. Therefore, we would like to investigate other types of processes with respect to their applicability to SFI.

Our future research would greatly benefit from carrying out numerous software projects from diverse domains. These developments, regardless of their size and complexity, are needed to provide more scientific significance to our current findings.

Once our research results are validated, we would like to investigate how the paradigm and its accompanying development process suit an industry setting. The projects described in this thesis produced usable software of good quality, but were carried out in an academic environment. These developments were oriented towards achieving research results instead of creating business value, the most important factor in the industry. Furthermore, the academic setting did not introduce practices, techniques and tools commonly found in organisations. We are confident that the paradigm can be adapted to suit these settings. However, we need to determine what elements of SFI need to be changed and specify the details of such modifications.

Stepwise Feature Introduction should not be considered as a generic remedy to all problems in all kinds of developments. Its use during software construction or reengineering should be preceded with a careful analysis of the drawbacks and the benefits. Furthermore, in each case the paradigm should be adjusted to match the requirements and the constraints of the system being developed, its stakeholders and the development team. Provided that these conditions are satisfied, we are confident that the paradigm of Stepwise Feature Introduction positively affects the quality of the produced software and can be successfully applied to development of large-scale, complex systems.

Appendices

1. Survey: Evaluation of *BioImageXD2* development process

The survey was carried after the development of the software has finished. The development team were asked 27 questions divided into 6 groups.

1.1. About you

1. What was your role (roles) in the development process?
    a. Programmer
    b. Designer
    c. Customer
    d. End user
2. What was your experience at the beginning of the BXD2 project, with respect to your role in the project?
    a. Poor
    b. Below average
    c. Average
    d. Good
    e. Excellent
3. For how many months have you worked on the project?

1.2. Project setting and complexity

4. How would you rate the complexity of the project, based on your experience and knowledge of similar software projects?
    a. Very simple
    b. Simple
    c. Not too simple, not too complex
    d. Complex
    e. Very complex
5. How often did you use the following development tools? (constantly, very often, seldom, very rare, not at all)
    a. Issue tracking system
    b. Project wiki
    c. Version control
6. Please rate the usefulness of the tools used during the development. (very useful, somewhat useful, not useful, have not used the tool)
    a. Issue tracking system

    b. Project wiki
    c. Version control
7. On scale from 1 (lowest) to 10 (highest) how would you rate the competence of the development team?

1.3. About the development process

8. How would you rate the development process and its suitability to the project?
    a. Very suitable
    b. Somewhat suitable
    c. Barely suitable
    d. Not suitable
9. How would you rate the development process, with respect to…? (no problems at all; minor problems, quickly resolved; major problems, took time to resolve; major problems, never resolved)
    a. Your understanding of the process
    b. You following the process
    c. Adapting the process to your needs
10. How often did you participate in the following meetings? (always, from time to time, rarely participated, never participated, not applicable)
    a. Daily scrum
    b. Sprint planning
    c. Sprint review
11. How would you rate the usefulness of each type of meeting? (very useful, somewhat useful, not useful)
    a. Daily scrum
    b. Sprint planning
    c. Sprint review
12. Do you think that the development process used with BXD2 can be used successfully when applied to a different project?
    a. Yes
    b. No
    c. Maybe

1.4. Design and implementation

13. At the beginning of the BXD2 project, how familiar you were with the following concepts? (no knowledge, poor, below average, average, good, excellent)
    a. Object-oriented programming
    b. Object-oriented design
    c. Agile development
    d. Java programming language
    e. Image processing methods and techniques
    f. Stepwise Feature Introduction
    g. Digital microscopy
14. How would you rate your knowledge about the architecture of the system?
    a. No knowledge
    b. Some knowledge
    c. Quite good knowledge
    d. Very good knowledge
15. To what extent did the design of the system take the following into account? (largest possible, important consideration, sometimes, seldom, never, do not know)
    a. Design for extensibility
    b. Design for performance
    c. Design for usability
    d. Design for maintainability
    e. Design for modularity
    f. Design for code reusability
16. To what extend did the implementation of the system take the following into account? (largest possible, important consideration, sometimes, seldom, never, do not know)
    a. Code for extensibility
    b. Code for performance
    c. Code for usability
    d. Code for maintainability
    e. Code for modularity
    f. Code for code reuse

17. Do you think that the layered architecture, followed by the stepwise introduction of features, can be of any use in other software projects? (yes, no, maybe)
    a. In smaller, less complex projects
    b. In projects of similar size and complexity
    c. In larger, more complex projects
18. On a scale from 1 (lowest) to 10 (highest) how would you rate the suitability of the architecture and the design to the project?
19. On a scale from 1 (lowest) to 10 (highest) how would you rate the suitability of the architecture and the design to the development process?

1.5. Comparison with previous development

20. Have you participated in the development of the Python version of *BioImageXD* at Åbo Akademi? If your answer to this question is No, please proceed to the next section.
    a. Yes
    b. No
21. Compared with the previous development, how would you rate the following? (much worse, worse, no change, better, much better, not applicable)
    a. Coding
    b. Designing
    c. Testing
    d. The software you built
    e. Your understanding of the project
22. Compared with the previous development, how would you rate the following? (much worse, worse, no change, better, much better, not applicable)
    a. Communication between the customer and the development team
    b. Organisation of the work
    c. Adaptation of the development process to the situation
    d. The overall satisfaction from the project

1.6. Concluding remarks

23. What, in your opinion, went good during the development?
24. What, in your opinion, went bad?
25. What surprised you during the development, with respect to the development process and the software?
26. What should be improved in the development process for future projects?
27. If there are things related to the project and the development that have not been covered by this survey, please, write them here.

2.Survey analysis

In this Appendix we present the most important findings of the survey that was carried after the development of *BioImageXD2* finished. The questions were detailed enough for the survey to act as a replacement for face-to-face interviews with the developers. The conclusions from the survey should contribute to improving the development process and increasing the possibility of its future reuse.

2.1. Personal information

The survey started with questions about personal information and prior experience. In the first question the respondents were asked to identify their role in the project. The distribution of answers is shown in Figure 65.

The development process was an adaptation of Scrum, which relies on cross-functional development team. The survey results confirm that this was also the case with *BioImageXD2*.

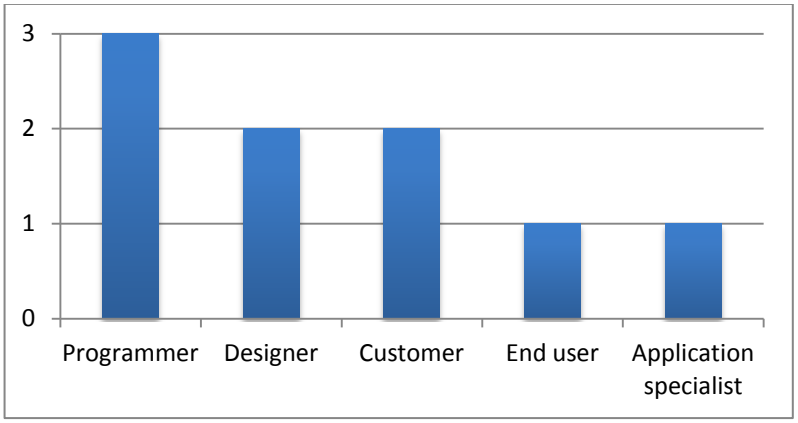


Figure 65. Roles of team members in the development process.

Figure 66 lists the experience of respondents prior to the project, with respect to their roles. The Product Owner is a specialist in the area of digital microscopy. The Team, on the other hand, consisted of computer science and computer engineering students that participated in similar, but smaller projects in the past; hence the overall experience of the programmers can be rated as good.

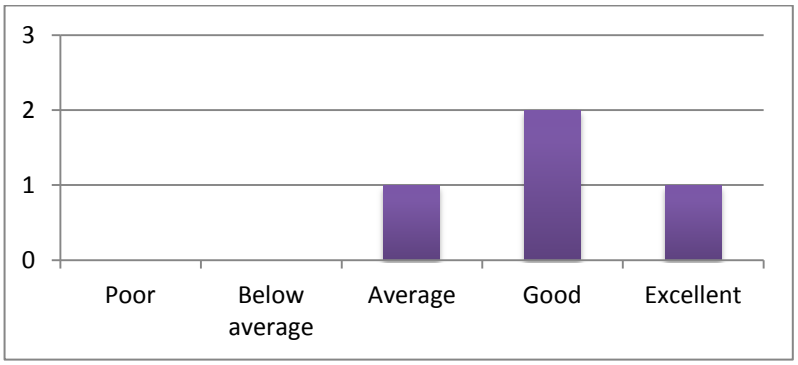


Figure 66. Experience of team members prior to the project.

The respondents were also asked about their knowledge of certain technologies used or needed during the development. The results are shown in Figure 67. The results provide a broader context for the analysis of the remaining questions.

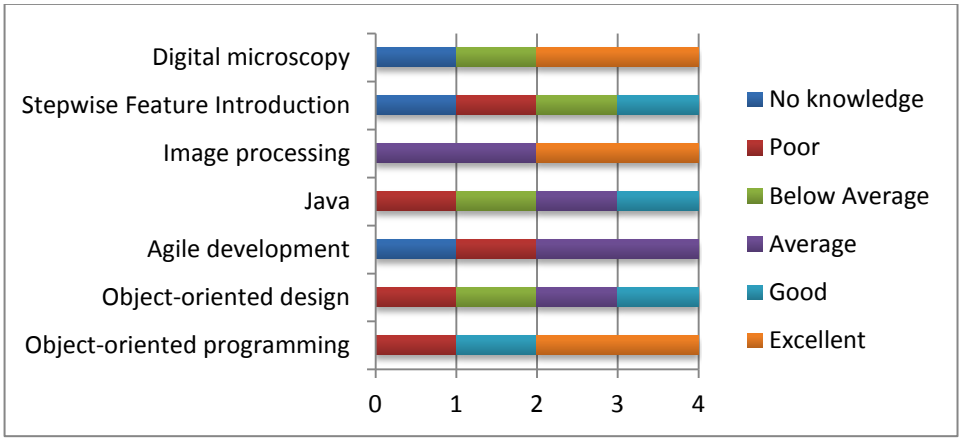


Figure 67. Knowledge of development technologies.

2.2. Project complexity

The complexity of the project was the focus of the second part of the survey. The respondents rated the project as rather complex, as shown in Figure 68. As mentioned previously, the survey was carried out once the project finished. Therefore, the Team and the Product Owner were aware of the difficulties that arose during the development.

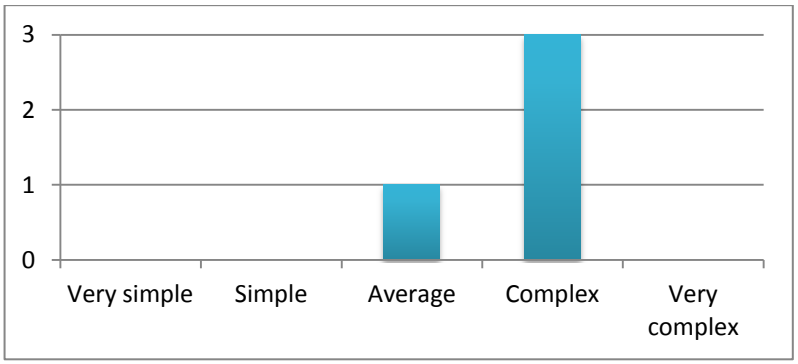


Figure 68. Project complexity.

Once the survey was completed, the developers were asked an additional question to identify the most complex features. The three-dimensional visualisation and image segmentation were unanimously chosen. This selection is further supported with the code metrics presented in the thesis.

2.3. The development process

The questions in the third part of the survey were related to the development process. The suitability of the process to the development was

rated positively, as presented in Figure 69. The one outstanding answer was given by the Product Owner, which implies that certain improvements related to this role must be made in future.

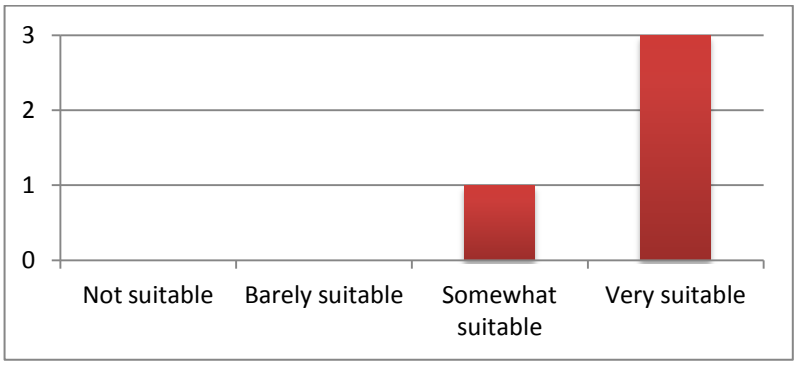


Figure 69. The suitability of the development process.

The subsequent question focused on the perception of the process by the developers. More specifically, we were interested in how easy it is for a developer to understand, follow and adapt the process. The results are given in Figure 70. Again, the answers are generally in favour of the process.

The major problems were raised by the Product Owner and one Team member. The Product Owner had major problems in adapting the process, which further indicates improvements to this role. The problem in understanding the process, reported by one of the Team members, was also resolved by providing the programmer with more information about Stepwise Feature Introduction and agile development methods.

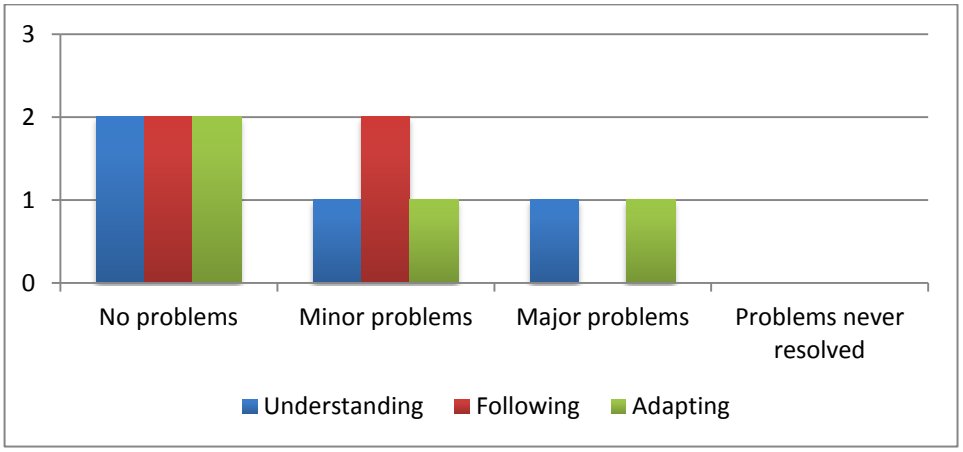


Figure 70. Understanding, following and adapting the development process.

The usefulness of different kinds of meetings was also evaluated, as shown in Figure 71. The feature review meetings were positively assessed by all

respondents. The usefulness of the other two kinds of meetings was graded similarly high. The results allow us to conclude that all of the meetings have been organised properly and were needed in the process.

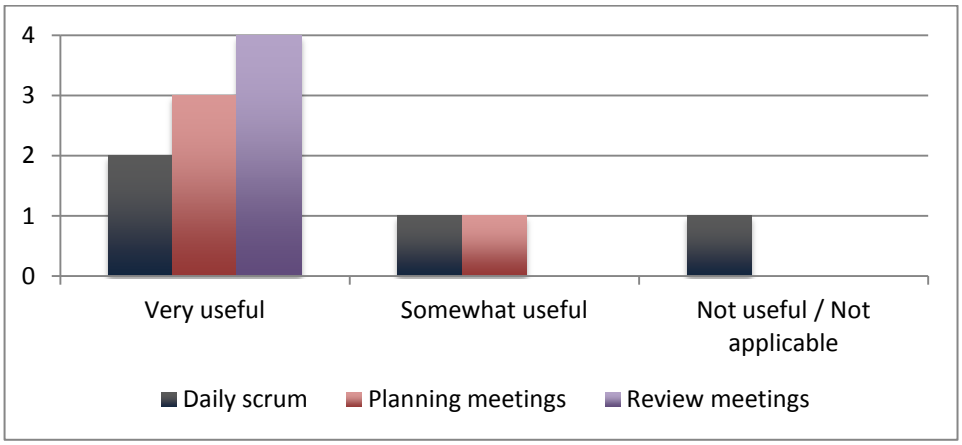


Figure 71. Usefulness of meetings in the development process.

The overall opinion about the development process is positive. From the developers perception it was straightforward to understand and follow, and suitable for the developed software. The possible improvements regard the role of the Product Owner, which should be adjusted to allow more flexibility, control and information exchange. Our conclusions were further reflected in the final question of this section, in which the Product Owner and the Team unanimously agreed that the development process can be successfully applied to a different project.

2.4. Design and implementation

The fourth section of the survey focused on the design and the implementation of the system. We asked the Team and the Product Owner to which extent the essential quality attributes of the system were taken into consideration during the development. The results are listed in Figure 72; the upper row for each characteristic concerns the design, whereas the lower is about the implementation.

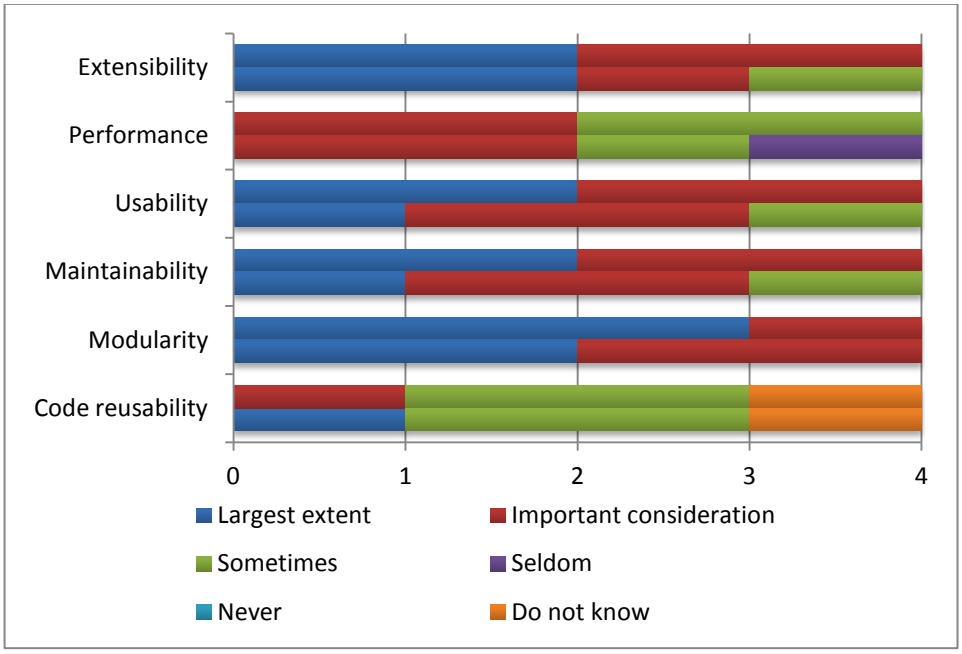


Figure 72. Characteristics of the design and the implementation.

The goal of the development was to produce a software system that is modular, maintainable, reusable and extendable. Based on the survey we can state that the Team and the Product Owner consider the design and the implementation of *BioImageXD2* to have all the required quality attributes. This subjective perception of the developers supports the results we obtained with code measurements.

The suitability of the architecture, with respect to the goals of the project and its development process has also been investigated, as shown in Figure 73. The respondents were to grade the suitability on the scale from 1 (least suitable) to 10 (best suitable). The responses given by the Product Owner are noticeably lower than the ones provided by the development team. This result confirms our initial findings and indicates that the role of the Product Owner must be reorganised.

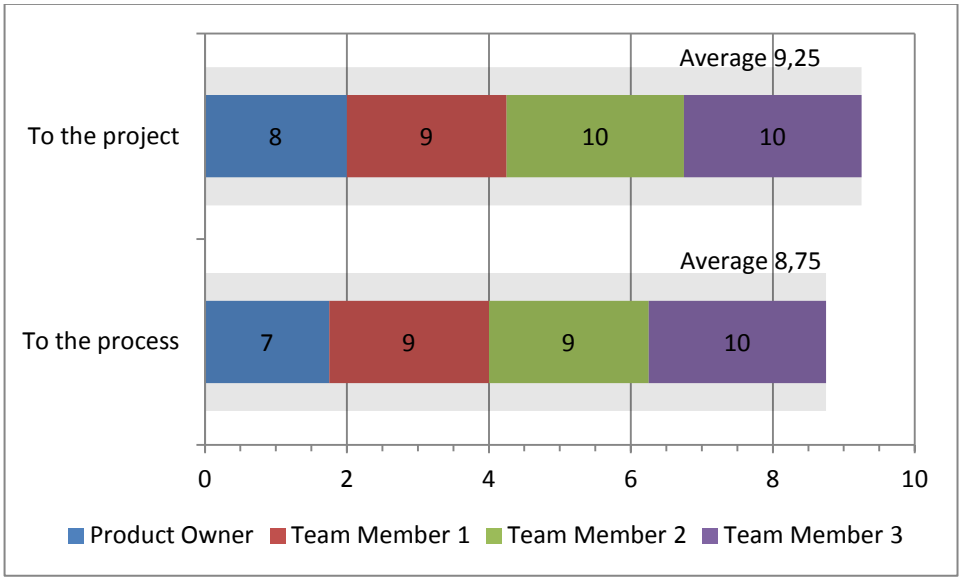


Figure 73. The suitability of the architecture to the project and to the process.

The overall suitability of the architecture was rated positively. Furthermore, the respondents consider Stepwise Feature Introduction an approach that can be reused in other projects, as shown in Figure 74. The paradigm is seen as an optimal choice for projects of similar or less complexity than *BioImageXD2*. The respondents were not convinced whether such approach is suitable for larger projects.

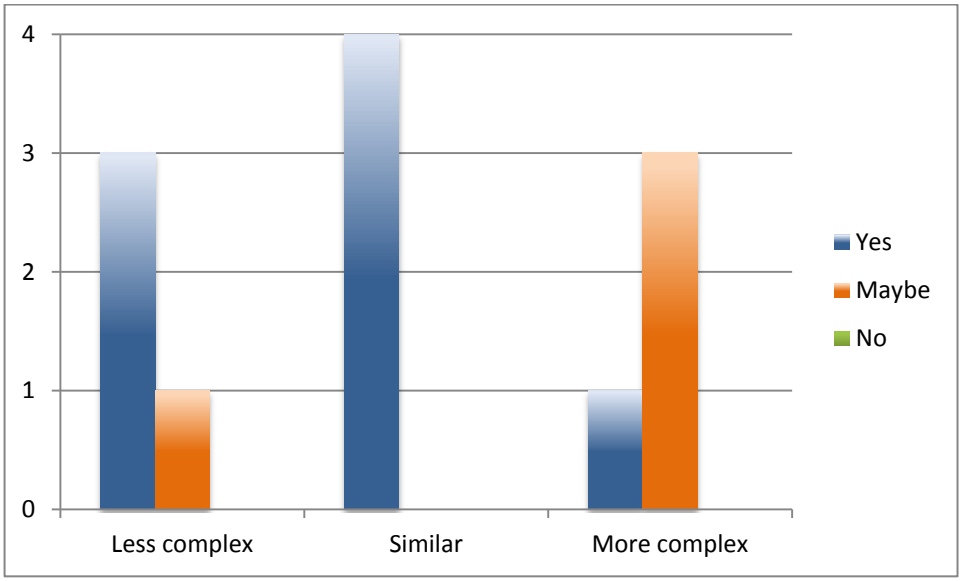


Figure 74. Possibility of reusing Stepwise Feature Introduction in other projects.

2.5. Comparison with previous development

The final part of the survey was aimed at comparing the development of *BioImageXD2* with its predecessor. Three of the respondents (the Product Owner and two Team members) answered the questions in this section, as they were directly involved in the development of the previous version of the software.

The results are gathered in Figure 75. It can be clearly seen that the newly developed version is perceived better in any aspect. However, the overall satisfaction of the development process has not changed for the Product Owner. Interestingly, all other aspects of the new version are seen by the Product Owner as better or much better than in the previous development, which matches the perception of the programming team. Such answer follows our other observations from the survey and indicates that the role of the Product Owner should be improved. The positive answers in the other areas lead to the conclusion that the development process was successful and produced software that is better, than its previous version.

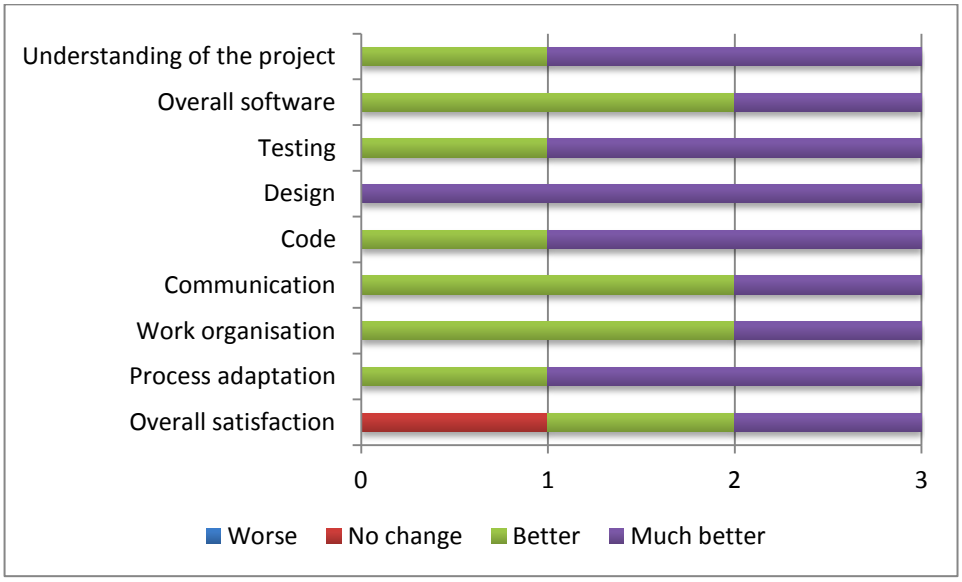


Figure 75. Comparison with the development of the previous version.

3.Listings

Listing 1. Explicit type checking caused by static typing in Java.

```
01. public class ServerRoom extends SingleGameRoom {
02.  /**
03.   * Submits the game. Queries before submission.
04.   * @param game Game to submit.
05.   */
06.  public void submitGame(Game game) {
07.    if(this.acceptGame(game)) super.submitGame(game);
08.    else this.doNotifyGameRejected(game);
09.  }
10.
11.  /**
12.   * Checks if the game can be submitted or not.
13.   * @param game Game.
14.   * @return If <b>true</b>, game should be submitted,
15.   *         otherwise rejected.
16.   */
17.  protected boolean acceptGame(Game game) {
18.    return game instanceof ServerGame;
19.  }
20. }
```

Listing 2. Unit test for class RectangleBoard from *ReThink*.

```
01. package test.pl.unforgiven.bge2.boards;
02.
03. import pl.unforgiven.bge2.boards.RectangleBoard;
04. import pl.unforgiven.bge2.board.Counter;
05. import pl.unforgiven.bge2.board.BoardPlayer;
06. import java.util.Random;
07. import java.lang.reflect.Method;
08. import org.junit.*;
09. import org.junit.runner.*;
10. import org.junit.runner.notification.Failure;
11. import static org.junit.Assert.*;
12.
13. /**
14.  * Test class for rectangle board.
15.  */
16. public class RectangleBoardTest {
17.
18.  protected static final Random RANDOM = new Random();
19.  private RectangleBoard board;
20.  private int rows, cols;
21.
22.  protected RectangleBoard getBoard() {
23.   return this.board;
24.  }
25.
26.  protected int getExpected(boolean rows) {
27.   return rows ? this.rows : this.cols;
28.  }
29.
30.  protected Counter getCounter(int playerNumber) {
31.   return new TestCounter((playerNumber%2)==0 ? "foo" : "bar");
32.  }
33.
34.  @Before public void setUp() {
35.   this.rows = RANDOM.nextInt(20)+1;
36.   this.cols = RANDOM.nextInt(20)+1;
37.   this.board = new RectangleBoard(this.cols, this.rows);
38.  }
39.
40.  @Test public void testGetDimensions() {
41.   assertEquals(this.getExpected(false),
42.                this.getBoard().getColumnCount());
43.   assertEquals(this.getExpected(true),
44.                this.getBoard().getRowCount());
45.   assertEquals(this.getExpected(false)*this.getExpected(true),
46.                this.getBoard().getSize());
47.  }
48.
49.  @Test public void testBoardEmpty() {
50.   for(int zmp1=0; zmp1<this.getBoard().getSize(); zmp1++)
51.    assertNull(this.getBoard().getCounter(zmp1));
52.   for(int zmp1=0; zmp1<this.getBoard().getColumnCount(); zmp1++)
```

```
53.    for(int zmp2=0; zmp2<this.getBoard().getRowCount(); zmp2++)
54.     assertNull(this.getBoard().getCounter(zmp1, zmp2));
55.  }
56.
57.  @Test public void testSetting()
58.            throws NoSuchMethodException, IllegalAccessException,
59.                   java.lang.reflect.InvocationTargetException {
60.   int col = RANDOM.nextInt(this.getBoard().getColumnCount());
61.   int row = RANDOM.nextInt(this.getBoard().getRowCount());
62.   Counter ctr = this.getCounter(1);
63.   // reflection executes protected method outside of class
64.   Method m = this.getBoard().getClass().
65.              getDeclaredMethod("setCounter",
66.              new Class[]{int.class, int.class, Counter.class});
67.   m.setAccessible(true);
68.   m.invoke(this.getBoard(), new Object[] {col, row, ctr});
69.   assertEquals(ctr, this.getBoard().getCounter(col, row));
70.   Counter ctr2 = this.getCounter(2);
71.   m.invoke(this.getBoard(), new Object[] {col, row, ctr2});
72.   assertEquals(ctr2, this.getBoard().getCounter(col, row));
73.  }
74. }
75. // Test runner output: 3 tests passed.
```

Listing 3. Unit test for RethinkBoard from *ReThink*.

```
01. package test.pl.rethink.base;
02.
03. import test.pl.unforgiven.bge2.boards.RectangleBoardTest;
04. import pl.unforgiven.bge2.boards.RectangleBoard;
05. import pl.unforgiven.bge2.board.Counter;
06. import pl.rethink.base.*;
07. import java.lang.reflect.Method;
08. import org.junit.Test;
09. import org.junit.runner.*;
10. import org.junit.runner.notification.Failure;
11. import static org.junit.Assert.*;
12.
13. /**
14.  * Tests for RethinkBoard.
15.  */
16. public class RethinkBoardTest extends RectangleBoardTest {
17.
18.  private RethinkPlayer p1 = RethinkPlayer.getPlayer("foo", 100);
19.  private RethinkPlayer p2 = RethinkPlayer.getPlayer("bar", 100);
20.  private RethinkBoard board = new RethinkBoard(6, 6);
21.
22.  protected RectangleBoard getBoard() {
23.   return this.board;
24.  }
25.
26.  protected int getExpected(boolean rows) {
27.   return 6;
28.  }
29.
30.  protected Counter getCounter(int playerNumber) {
31.   return (playerNumber%2)==0 ? this.p1 : this.p2;
32.  }
33.
34.  @Test public void testPushUp()
35.            throws NoSuchMethodException, IllegalAccessException,
36.                   java.lang.reflect.InvocationTargetException {
37.   int col = RANDOM.nextInt(6)+1;
38.   Method m = this.getBoard().getClass().
39.                   getDeclaredMethod("pushColumnUp",
40.                    new Class[]{int.class, RethinkPlayer.class});
41.   m.setAccessible(true);
42.   m.invoke(this.getBoard(),
43.            new Object[] {col, this.getCounter(1)});
44.   assertEquals(this.getCounter(1),
45.                this.getBoard().getCounter(col, 5));
46.   m.invoke(this.getBoard(),
47.            new Object[] {col, this.getCounter(2)});
48.   assertEquals(this.getCounter(2),
49.                this.getBoard().getCounter(col, 5));
50.   assertEquals(this.getCounter(1),
51.                this.getBoard().getCounter(col, 4));
52.   for(int zmp1=0; zmp1<6; zmp1++)
```

```
53.     m.invoke(this.getBoard(),
54.              new Object[] {col, this.getCounter(1)});
55.   for(int zmp1=0; zmp1<6; zmp1++)
56.     for(int zmp2=0; zmp2<6; zmp2++)
57.       if(zmp1==col) assertEquals(this.getCounter(1),
58.                         this.getBoard().getCounter(zmp1, zmp2));
59.       else assertNull(this.getBoard().getCounter(zmp1, zmp2));
60.  }
61.
62.  @Test public void testPushLeft()
63.            throws NoSuchMethodException, IllegalAccessException,
64.                   java.lang.reflect.InvocationTargetException {
65.   int row = RANDOM.nextInt(6)+1;
66.   Method m = this.getBoard().getClass().
67.                   getDeclaredMethod("pushRowLeft",
68.                    new Class[]{int.class, RethinkPlayer.class});
69.   m.setAccessible(true);
70.   m.invoke(this.getBoard(),
71.            new Object[] {row, this.getCounter(1)});
72.   assertEquals(this.getCounter(1),
73.                this.getBoard().getCounter(5, row));
74.   m.invoke(this.getBoard(),
75.            new Object[] {row, this.getCounter(2)});
76.   assertEquals(this.getCounter(2),
77.                this.getBoard().getCounter(5, row));
78.   assertEquals(this.getCounter(1),
79.                this.getBoard().getCounter(4, row));
80.   for(int zmp1=0; zmp1<6; zmp1++)
81.     m.invoke(this.getBoard(),
82.              new Object[] {row, this.getCounter(1)});
83.   for(int zmp1=0; zmp1<6; zmp1++)
84.     for(int zmp2=0; zmp2<6; zmp2++)
85.       if(zmp2==row) assertEquals(this.getCounter(1),
86.                         this.getBoard().getCounter(zmp1, zmp2));
87.       else assertNull(this.getBoard().getCounter(zmp1, zmp2));
88.  }
89. }
90. // Test runner output: 5 tests passed.
```

4. Quality report for *BioImageXD2*

4.1. Metrics Summary

Metric	Value
Number of Libraries	39
Number of Packages	34
Number of Top Level Classes	333
Average Number of Top Level Classes per Package	9.79
Average Number of Member Classes per Class	0.08
Average Number of Methods per Class	6.21
Average Number of Fields per Class	2.08
Estimated Lines of Code	27601
Estimated Lines of Code per Top Level Class	82.89
Average Cyclomatic Complexity	1.40
Fat for Library Dependencies	125
Fat for Flat Package Dependencies	137
Fat for Top Level Class Dependencies	1890
Tangled for Library Dependencies	0%
Average Component Dependency between Libraries	13.23%
Average Component Dependency between Packages	19.96%
Average Component Dependency between Units	9.72%
Average Distance	-0.19
Average Absolute Distance	0.27
Average Weighted Methods per Class	8.70
Average Depth of Inheritance Tree	1.35
Average Number of Children	0.60
Average Coupling between Objects	3.84
Average Response for a Class	11.01
Average Lack of Cohesion in Methods	30.94

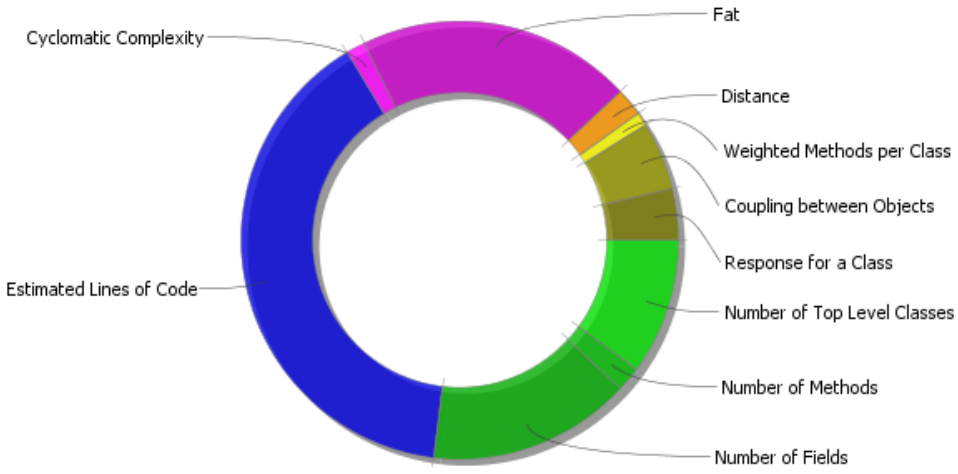
4.2. Top Violations (20 of 128)

Artifact	Metric	Value
bio.modules.processes	Units	42
bio.modules.visualisations.Visualization3D	ELOC	1891
bio.modules.visualisations.Visualization3D	Fat	356
bio.modules.visualisations.Visualization3D	Fields	111

bio.main.BioWindow	ELOC	874
bio.modules.processes.HSBColouringProcess	ELOC	723
bio.modules.processes.ColouringProcess	ELOC	720
bio.modules.visualisations.Visualization3D	RFC	151
bio.modules.visualisations.Visualization3D.makeSettingsContents(...)	ELOC	1042
bio.modules.visualisations.Visualization3D	CBO	31
bio.modules.visualisations.Gallery	ELOC	657
bio.modules.processes.HSBColouringProcess	Fat	140
bio.modules.processes.ColouringProcess	Fat	156
bio.gui.adapters.swing.SwingXYCanvas	ELOC	656
bio.gui.adapters.swing.SwingXYCanvas	Fat	198
bio.modules.processes.HSBColouringTask	ELOC	490
bio.modules.processes.ColouringProcess	Fields	36
bio.main.BioWindow	Fat	96
bio.modules.visualisations.animator.SplineEditor	ELOC	469
bio.modules.processes.HSBColouringProcess	Fields	32

4.3. Pollution Chart

Pollution 1.21



4.4. Violations by Metric

Number of Top Level Classes

Artifact	Value
bio.modules.processes	42

Number of Methods

Artifact	Value
bio.gui.adapters.swing.SwingXYCanvas	84
bio.gui.adapters.swing.SwingListSpinner	52
bio.gui.adapters.swing.SwingListBox	51
bio.util.modules.Config3D	52
bio.gui.components.XYCanvas	57

Number of Fields

Artifact	Value
bio.modules.visualisations.Visualization3D	111
bio.modules.processes.ColouringProcess	36
bio.modules.processes.HSBColouringProcess	32
bio.main.BioWindow	28
bio.modules.processes.HSBColouringTask	36
bio.modules.visualisations.Gallery	26
bio.gui.adapters.swing.SwingXYCanvas	21
bio.modules.visualisations.animator.SplineEditor	21
bio.modules.processes.OtsuSegmentationProcess	22
bio.util.modules.Config3D	26

Estimated Lines of Code

Artifact	Value
bio.modules.visualisations.Visualization3D	1891
bio.main.BioWindow	874
bio.modules.processes.HSBColouringProcess	723
bio.modules.processes.ColouringProcess	720

bio.modules.visualisations.Visualization3D.makeSettingsContents(...)	1042
bio.modules.visualisations.Gallery	657
bio.gui.adapters.swing.SwingXYCanvas	656
bio.modules.processes.HSBColouringTask	490
bio.modules.visualisations.animator.SplineEditor	469
bio.modules.processes.HSBColouringProcess.makeSettingsContents(...)	423
bio.modules.visualisations.Orthogonal	413
bio.modules.processes.ColouringProcess.makeSettingsContents(...)	395
bio.modules.processes.OtsuSegmentationProcess	376
bio.modules.processes.HistogramTask	377
bio.gui.adapters.swing.SwingTable	383
bio.main.BioWindow.getWindow()	457
bio.modules.processes.InterpolationProcess	345
bio.gui.adapters.swing.SwingSplitPanel	351
bio.modules.processes.HistogramTask.calculate(...)	154
bio.modules.processes.OtsuSegmentationProcess.makeSettingsContents(...)	259
bio.modules.processes.InterpolationProcess.makeSettingsContents(...)	304
bio.modules.visualisations.Visualization3D.updateView()	144
bio.modules.visualisations.animator.AnimatorVisualizer	310
bio.modules.visualisations.Visualization3D.volumeRender3D(...)	115
bio.modules.processes.HistogramProcess.makeSettingsContents(...)	171
bio.modules.visualisations.Orthogonal.updateView()	113
bio.gui.adapters.swing.SwingListBox	303
bio.gui.adapters.swing.SwingListSpinner	303
bio.main.SettingsDialog.getSettingsDialog(...)	126
bio.modules.processes.AnisotropicNoiseFilterProcess.makeSettingsContents(...)	180
bio.modules.visualisations.Gallery.makeDisplaySettingsPage(...)	125
bio.modules.processes.CropProcess.makeSettingsContents(...)	174
bio.modules.processes.HistogramTask.calculate(...)	102
bio.modules.visualisations.Gallery.makeViewSettingsPage(...)	104

bio.java.util.PackageClassLoader.getClassesForPackage(...)	93
bio.modules.processes.ObjectSeparationProcess.makeSettingsContents(...)	132
bio.vtk.gui.adapters.swing.SwingVTKPanel.VTKPanel.mouseDragged(...)	83
bio.modules.processes.SimpleAdjustProcess.makeSettingsContents(...)	136
bio.modules.processes.AdjustProcess.makeSettingsContents(...)	115
bio.modules.visualisations.Gallery.updateView()	82
bio.modules.processes.MorphologicalWatershedProcess.makeSettingsContents(...)	128
bio.modules.processes.SimpleColouringProcess.makeSettingsContents(...)	124
bio.modules.visualisations.animator.VideoEncoder.getCommandLine(...)	75
bio.modules.visualisations.animator.VideoEncoder.()	73
bio.modules.processes.ResizeProcess.makeSettingsContents(...)	140
bio.modules.visualisations.Orthogonal.makeSettingsContents(...)	88
bio.modules.visualisations.Visualization3D.surfaceRender3D(...)	75
bio.modules.visualisations.Gallery.arrangeDrawingSpace(...)	67
bio.modules.visualisations.animator.AnimatorVisualizer.setView(...)	66
bio.modules.processes.ColocalizationProcess.updateThresholdGUI()	87
bio.math.algorithms.LineClip.clip(...)	64
bio.modules.processes.OtsuSegmentationTask.calculateHistogram(...)	64
bio.modules.processes.HSBColouringTask.createHSBColorLookupTable(...)	66
bio.modules.filereaders.VTKReader.doLoad(...)	64
bio.modules.processes.ThresholdNoiseFilterTask.calculateHistogram(...)	63
bio.xml.XMLNodeMapper.setAttribute(...)	63
bio.modules.processes.AdjustTask.applyNativeTransformation(...)	65
bio.modules.processes.InterpolationTask.applyNativeTransformation(...)	64
bio.modules.visualisations.animator.SplineEditor.getCameraPosition(...)	64
bio.modules.processes.LabellingProcess.makeSettingsContents(...)	83
bio.main.BioWindow.getFileReaderHandler(...)	73
bio.gui.adapters.swing.SwingXYCanvas.(...)	83
bio.modules.processes.ColocalizationProcess.makeSettingsContents(...)	60
bio.main.BioWindow.getVisualisationHandler(...)	60
bio.modules.visualisations.Orthogonal.makeDrawingContents(...)	84

Cyclomatic Complexity

Artifact	Value
bio.modules.visualisations.Visualization3D.updateView()	19
bio.modules.processes.HistogramTask.calculate(...)	17
bio.java.util.PackageClassLoader.getClassesForPackage(...)	22
bio.modules.processes.HistogramTask.calculate(...)	18
bio.modules.processes.AdjustTask.applyNativeTransformation(...)	17
bio.modules.visualisations.animator.SplineEditor.getCameraPosition(...)	17
bio.modules.processes.InterpolationTask.applyNativeTransformation(...)	16
bio.modules.processes.HSBColouringTask.createHSBColorLookupTable(...)	15

Fat

Artifact	Value
bio.modules.visualisations.Visualization3D	356
bio.modules.processes.HSBColouringProcess	140
bio.modules.processes.ColouringProcess	156
bio.gui.adapters.swing.SwingXYCanvas	198
bio.main.BioWindow	96
bio.modules.visualisations.Gallery	109
bio.modules.visualisations.animator.SplineEditor	76
bio.modules.processes.OtsuSegmentationProcess	75
bio.modules.visualisations.Orthogonal	63
bio.gui.adapters.swing.SwingTable	78
bio.vtk.gui.adapters.swing.SwingVTKPanel	73
bio.util.modules.Config3D	100
bio.modules.visualisations.animator.VideoEncoder	69
bio.modules.processes.CropTask	77
bio.gui.adapters.swing.SwingListBox	66
bio.gui.adapters.swing.SwingRadioList	63
bio.gui.adapters.swing.SwingDialog	65
bio.gui.components	66
bio.modules.processes.ResizeTask	67

Distance

Artifact	Value
bio.gui.adapters.swing.helpers	-0.56
bio.gui.helpers	-0.58
bio.config.xml	-0.75
bio.math	-0.83
bio.java.util	-1
bio.base.colormodels	-0.60
bio.gui.events	-0.52
bio.util.image	-1

Weighted Methods per Class

Artifact	Value
bio.gui.adapters.swing.SwingXYCanvas	113
bio.gui.adapters.swing.SwingSplitPanel	106

Coupling between Objects

Artifact	Value
bio.modules.visualisations.Visualization3D	31
bio.main.BioWindow	46
bio.modules.processes.ColouringProcess	26
bio.modules.visualisations.Gallery	34
bio.gui.components.Factory	28
bio.gui.components.ComponentContainer	27

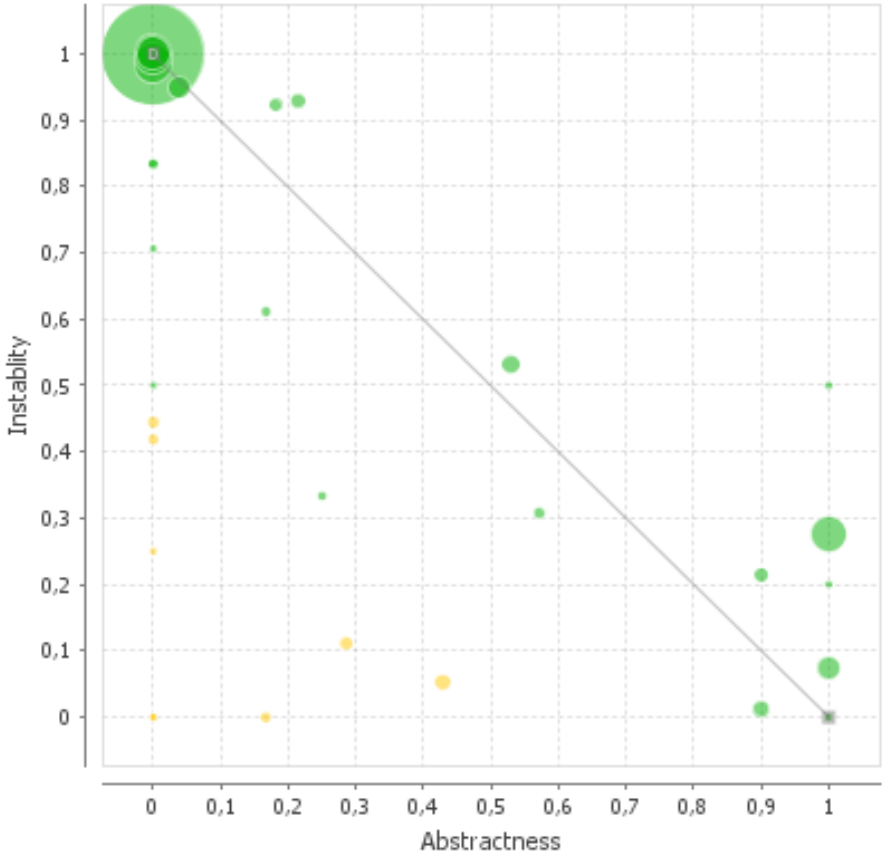
Response for a Class

Artifact	Value
bio.modules.visualisations.Visualization3D	151
bio.modules.visualisations.Gallery	110
bio.gui.adapters.swing.SwingXYCanvas	109

4.5. Design Tangles

There are no design tangles.

4.6. Package Distance Chart

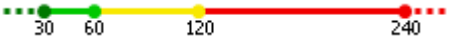
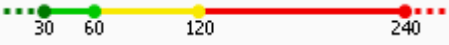


4.7. Metric Ratings

Count Metrics

Metric	Rating	Linear
Number of Top Level Classes	20 40 60 80	✓
Number of Methods	25 50 100 200	✓
Number of Fields	10 20 40 80	✓
Estimated Lines of Code	200 300 400 500	✓
Estimated Lines of Code	30 60 120 240	✓

Complexity Metrics

Metric	Rating	Linear
Cyclomatic Complexity		✓
Fat		✓
Fat		✓
Fat		✓
Tangled		✓
Tangled for Library Dependencies		✓
Average Component Dependency between Libraries		✓
Average Component Dependency between Packages		✓

Robert C. Martin Metrics

Metric	Rating	Linear
Distance		✓
Average Absolute Distance		✓

Chidamber & Kemerer Metrics

Metric	Rating	Linear
Weighted Methods per Class		✓
Depth of Inheritance Tree		✓
Average Depth of Inheritance Tree		✓
Coupling between Objects		✓
Response for a Class		✓

Bibliography

[1] J.-R. Abrial, *The B-book: assigning programs to meanings*. Cambridge: Cambridge University Press, 1996.

[2] J.-R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen, "The B-method", *Lecture Notes in Computer Science*, vol. 552, pp. 398-405, 1991.

[3] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental Regression Testing", in *Proceedings of Conference on Software Maintenance*, Montreal, Canada, 1993, pp. 384-357.

[4] W. Ahrendt et al., "The KeY System: Integrating Object-Oriented Design and Formal Methods", in *Fundamental Approaches to Software Engineering*. Berlin: Springer Berlin / Heidelberg, 2002.

[5] ARiSA AB. Compendium of Software Quality. [Online]. http://www.arisa.se/compendium/node88.html

[6] Deborah J. Armstrong, "The Quarks of Object-Oriented Development", *Communications of the ACM*, vol. 49, no. 2, pp. 123-128, 2006.

[7] R.-J. Back, *On the Correctness of Refinement Steps in Program Development*. Helsinki: University of Helsinki, 1978.

[8] Ralph Johan Back, "Software Construction by Stepwise Feature Introduction", in *ZB 02: Proceedings of the 2nd International Conference of B and Z Users of Formal Specification and Development in Z and B*. Springer-Verlag, 2002.

[9] R.-J. Back, J. Eriksson, and L. Milovanov, "Using stepwise feature introduction in practice: an experience report", in *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2005)*, 2005, pp. 2-17.

[10] R.-J. Back and J. von Wright, *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[11] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte, "Verification of object-oriented programs with invariants", vol. 3, no. 6, 2004.

[12] V. R. Basili, "The experimental paradigm in software engineering", in *Experimental software engineering issues: critical assessment and future directives*. New York: Springer Lecture Notes in Computer Science 706, 1993.

[13] V. R. Basili, W. Selby, and D. H. Hutchents, "Experimentation in Software Engineering", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 7, pp. 733-743, July 1986.

[14] Kent Beck, *Extreme Programming Explained*. Addison-Wesley, 1999.

[15] Kent Beck, *Test-Driven Development by Example*. Addison-Wesley, 2003.

[16] Kent Beck et al. Agile Manifesto. [Online]. http://agilemanifesto.org

[17] Boris Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold, 1990.

[18] Herbert D. Benington, "Production of Large Computer Programs", *IEEE Annals of the History of Computing*, vol. 5, no. 4, pp. 350-361, 1983.

[19] Robert V. Binder, *Testing Object-Oriented Systems: Objects, Patterns, and Tools*. Addison-Wesley Professional, 1999.

[20] BioImageXD development team. BioImageXD. [Online]. http://www.bioimagexd.net

[21] Rex Black, *Managing the Testing Process*. Microsoft Press, 1999.

[22] Barry Boehm, "A Spiral Model of Software Development and Enhancement", *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 4, pp. 14-24, 1986.

[23] Barry Boehm and Richard Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*, 7th ed. Boston, MA: Addison-Wesley, 2004.

[24] Grady Booch, *Object-Oriented Design with Applications*, 3rd ed. Addison-Wesley, 2007.

[25] Chris Bourne, "Think!", *Sinclair User*, p. 62, 1986.

[26] Jonathan P. Bowen and Michael G. Hinchey, "Ten Commandments of Formal Methods", *Computer*, vol. 28, no. 4, pp. 56-63, 1995.

[27] F. P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering", *Computer Magazine*, 1987.

[28] Frederick Brooks, *The Mythical Man-Month*, 2nd ed. Addison-Wesley, 1975.

[29] R. W. Butler. NASA LaRC Formal Methods Program. [Online]. http://shemesh.larc.nasa.gov/fm/fm-what.html

[30] M. Butler. Refinement Calculus. [Online]. http://users.ecs.soton.ac.uk/mjb/refcalc-tut/home.html

[31] M. Butler. Refinement Calculus Tutorial. [Online]. http://users.ecs.soton.ac.uk/mjb/refcalc-tut/prognot.html

[32] M. Butler. Refinement Calculus Tutorial. [Online]. http://users.ecs.soton.ac.uk/mjb/refcalc-tut/laws.html

[33] M. Butler and C. Snook. UML-B. [Online]. http://users.ecs.soton.ac.uk/cfs/umlb.html

[34] Jon Byous, "Java technology: The early years", *Sun Developer Network*, 1998.

[35] Cambridge University Press. Cambridge Dictionary Online. [Online]. http://dictionary.cambridge.org/dictionary/british/feature_1

[36] Stuart Campbell, "The Your Sinclair Official All-Time Top 100 Spectrum Games", *Your Sinclair*, 1991.

[37] Jones Capers, *Applied Software Measurement*, Second Edition ed. McGraw Hill, 1996.

[38] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2", in *Formal Methods for Components and Objects - Lecture Notes in Computer Science 4111/2006*, 2006, pp. 342-363.

[39] Y. Cheon and G. T. Leavens, "A runtime assertion checker for the Java Modelling Language (JML)", in *Proceedings of the International Conference on Software Engineering Research and Practice*, 2002, pp. 322-328.

[40] S. R. Chidamber and Ch. F. Kemerer, "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493, 1994.

[41] Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford, "Documenting software architectures: views and beyond", in *Proceedings of the 25th International Conference on Software Engineering*, Washington, USA, 2003, pp. 740-741.

[42] P. Coad, E. Lefebvre, and J. De Luca, *Java Modeling in Color with UML: Enterprise Components and Process*. Prentice Hall International, 1999.

[43] Computer Science Laboratory, SRI International. PVS Specification and Verification System. [Online]. http://pvs.csl.sri.com

[44] Computerworld staff. Computerworld Online. [Online]. http://www.computerworld.com/s/article/100542/Computerworld_Development_Survey_gives_nod_to_C_?taxonomyId=011

[45] J. Crinnion, *Evolutionary Systems Development. A practical guide to the use of prototyping wihin a structured systems methodology*. New York: Plenum Press, 1991.

[46] Alan M. Davis, "Operational Prototyping: A New Development Approach", *IEEE Software*, no. 7, p. 71, 1992.

[47] Robin S. Davis, *Who's Sitting on Your Nest Egg?* Austin: Bridgeway Books, 2007.

[48] E. W. Dijkstra, *Notes on structured programming*. Academin Press, 1972.

[49] B. Eckel and B. Venners. Artima Developer. [Online]. http://www.artima.com/intv/prodperfP.html

[50] Eclipse Foundation. Open source community website. [Online]. http://www.eclipse.org

[51] Edgewall Software. The Trac Project. [Online]. http://trac.edgewall.org

[52] Holger Eichelberger, "Aesthetics of Class Diagrams", in *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis*, 2002, pp. 23-31.

[53] Encyclopædia Britannica. Encyclopædia Britannica Online. [Online]. http://www.britannica.com/EBchecked/topic/486323/quark

[54] J. W. E. Eriksson, *Tool-Supported Invariant-Based Programming*. Turku: Turku Centre for Computer Science (TUCS), 2010.

[55] Norman E. Fenton, *Software Metrics: A Rigorous Approach*, 2nd ed. London, UK: Chapman & Hall, Ltd., 1991.

[56] C. Flanagan et al., "Extended Static Checking for Java", in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, New York, 2002.

[57] D. Flanagan and Y. Matsumoto, *Ruby Programming Language*, 1st ed. O'Reilly Media, 2008.

[58] Matthew Ford. (2011, Nov.) Aspect-Oriented Programming in Ruby. [Online]. http://www.slideshare.net/deimos/aspect-orientated-programming-in-ruby

[59] K. Forsberg, H. Mooz, and H. Cotterman, *Visualizing Project Management*, 3rd ed. New York: John Wiley and Sons, 2005.

[60] Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modelling Language*. Pearson Education, 2004.

[61] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[62] D. Garlan and M. Shaw, "An Introduction to Software Architecture", in *Advances in Software Engineering*. New Jersey: World Scientific Publishing Company, 1993, vol. I.

[63] J. Gorman. WikiWikiWeb. [Online]. http://c2.com/cgi/wiki?FormalSpecification

[64] Paul Graham. Paul Graham Home Page. [Online]. http://www.paulgraham.com/web20.html

[65] Paul Graham. (2002, May) Paul Graham's Home Page. [Online]. http://www.paulgraham.com/icad.html

[66] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Pothermel, "An Empirical Study of Regression Test Selection Techniques", *ACM Transactions of Software Engineering and Methodology*, vol. 10, no. 2, pp. 184-208, 2001.

[67] J. V. Guttag et al., *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[68] A. Hall, "Seven Myths of Formal Methods", *IEEE Software*, vol. 1990, no. 9, pp. 11-19, 1990.

[69] B. Henderson-Sellers, *A Book of Object-Oriented Knowledge*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[70] S. Henry and M. Humphrey, *Comparison of an Object-Oriented Programming Language to a Procedural Programming Language for Effectiveness in Program Maintenance*. Balcksburg, Virginia: Virginia Polytechnic Institute, 1988.

[71] Alison Hjul, "Think", *Your Sinclair*, 1986.

[72] C. A. R. Hoare, "Proof of Correctness of Data Representations", *Acta Informatica*, vol. 1, pp. 271-281, 1972.

[73] Allen I. Holub. Holum Associates. [Online]. http://www.holub.com/goodies/uml

[74] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. New York: IEEE-SA Standards Board, 2008.

[75] ISO, *ISO/IEC 9126-1:2001*. ISO, 2001.

[76] S. M. Jamali, "Object Oriented Metrics (A Survey Approach)", Sharif University of Technology, Teheran, Iran, Course Paper 2006.

[77] Ralph. E Johnson and Brian Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, pp. 22-35, 1988.

[78] Gregor Kiczales et al., *Aspect-Oriented Programming*. Jyväskylä, Finland: Springer Lecture Notes in Computer Science 1241, Springer-Verlag, 1997.

[79] B. A. Kitchenham et al., "Preliminary Guidelines for Empirical Research in Software Engineering", 2001.

[80] Kitware, Inc. ITK Home Page. [Online]. http://www.itk.org

[81] Kitware, Inc. VTK Home Page. [Online]. http://www.vtk.org

[82] D. C. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On Regression Testing of Object-Oriented Programs", *Journal of Systems Software*, vol. 32, pp. 21-40, 1996.

[83] John Lakos, *Large-Scale C++ Software Design*. Addison-Wesley Professional, 1996.

[84] V. Le Hahn, K. Akif, Y. Le Traon, and J.-M. Jézéquel, "Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies", in *Lecture Notes in Computer Science*. Springer-Verlag, 2001, vol. 2072.

[85] G. Leavens. The Java Modelling Language (JML). [Online]. http://www.eecs.ucf.edu/~leavens/JML/

[86] G. T. Leavens and Y. Cheon, "Design by Contract with JML", Iowa State University, Ames, 2006.

[87] K. Rustan M. Leino and P. Müller, "Object Invariants in Dynamic Contexts", in *ECOOP 2004 - Object-Oriented Programming*. Berlin: Springer Berlin / Heidelberg, 2004.

[88] H. K. N. Leung and L. White, "A Study of Integration Testing and Software Regression at the Integration Level", in *Proceedings of Conference on Software Maintenance*, San Diego, USA, 1990, pp. 290-301.

[89] K. Lieberher, I. Holland, and A. Riel, "Object-Oriented Programming: An Objective Sense of Style", in *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, San Diego, the United States, 1988, pp. 323-334.

[90] Barbara Liskov, "Data abstraction and hierarchy", in *Addenum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications*, Orlando, Florida, the United States, 1987, pp. 17-14.

[91] Robert C. Martin. Object Mentor. [Online]. http://www.objectmentor.com/resources/articles/ocp.pdf

[92] Robert C. Martin. Object Mentor. [Online]. http://www.objectmentor.com/resources/articles/lsp.pdf

[93] Robert C. Martin. Object Mentor. [Online]. http://www.objectmentor.com/resources/articles/dip.pdf

[94] Robert C. Martin. Object Mentor. [Online]. http://www.objectmentor.com/publications/granularity.pdf

[95] Robert C. Martin. Object Mentor. [Online]. http://www.objectmentor.com/resources/articles/stability.pdf

[96] Robert C. Martin. Object Mentor. [Online]. http://www.objectmentor.com/publications/isp.pdf

[97] Robert C. Martin, "Object Oriented Design Quality Metrics: An Analysis of Dependencies", *ROAD*, vol. 2, 1995.

[98] Robert C. Martin. Principles Of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?SingleResponsibilityPrinciple

[99] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?OpenClosedPrinciple

[100] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple

[101] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?InterfaceSegregationPrinciple

[102] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?ReuseReleaseEquivalencePrinciple

[103] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?CommonClosurePrinciple

[104] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?CommonReusePrinciple

[105] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?CommonReusePrinciple

[106] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?StableDependenciesPrinciple

[107] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?StableAbstractionsPrinciple

[108] Robert C. Martin. Principles of Object Oriented Design. [Online]. http://c2.com/cgi/wiki?DependencyInversionPrinciple

[109] Robert C. Martin. WikiWikiWeb (Cunningham & Cunningham). [Online]. http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign

[110] T. J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308-320, 1976.

[111] Atif M. Memon, "GUI Testing: Pitfalls and Process", *IEEE Computer*, no. 8, pp. 87-88, 2002.

[112] Atif M. Memon, "GUI Testing: Pitfalls and Process", *IEEE Computer*, vol. 8, pp. 87-88, 2002.

[113] G. Meszaros, "Agile Regression Testing Using Record & Playback", in *Proceedings of Conference on Object Oriented Programming System Languages and Applications*, Anaheim, USA, 2003, pp. 353-360.

[114] Bertrand Meyer, *Object-Oriented Software Construction*. Prentice Hall, 2000.

[115] Microsoft Patterns and Practices Team, *Microsoft Application Architecture Guide*, 2nd ed. Microsoft Press, 2009.

[116] Luka Milovanov, *Agile Software Development in an Academic Environment*. Turku: TUCS Dissertations, 2006.

[117] G. Myers, *Software reliability: Principles and practices*. New York: Wiley, 1976.

[118] J. D. Naumann and A. M. Jenkins, "Prototyping: The New Paradigm for Systems Development", *MIS Quarterly*, vol. 6, no. 3, pp. 29-44, 1982.

[119] J. Nielsen, *Usability Engineering*. Morgan Kaufmann Publishers, 1994.

[120] Peter Norvig. (1998, Mar.) Peter@Norvig.com.

[121] Conrad Nutschan. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/wiki/File:Spiral_model_(Boehm,_1988).png

[122] Kristen Nygaard and Ole-Johan Dahl, "The Development of the SIMULA Languages", *ACM SIGPLAN Notices*, vol. 13, no. 8, pp. 245-272, 1978.

[123] Object Management Group. UML. [Online]. http://www.uml.org

[124] Object Management Group, *UML 2.3 Specification*. Object Management Group, 2010.

[125] Odysseus Software GmbH. stan4j.com. [Online]. http://stan4j.com

[126] Odysseus Software, GmbH, "STAN Metric Definitions", in *STAN Reference*. Odysseus Software, GmbH, 2011.

[127] Michael Olan, "Unit testing: test early, test often", *Journal of Computing Sciences in Colleges*, vol. 19, no. 2, pp. 319-328, 2003.

[128] Marta Olszewska (Pląska), *On the Impact of Rigorous Approaches on the Quality of Development*. Turku, Finland: TUCS (Turku Centre for Computer Science), 2011.

[129] Open Source Community. Computer Language Benchmarks Game. [Online]. http://shootout.alioth.debian.org/u64q/benchmark.php?test=all&lang=java&lang2=gpp

[130] Oracle, Inc. Oracle Technology Network: Java. [Online]. http://www.oracle.com/technetwork/java/codeconventions-135099.html

[131] Oracle, Inc. The Java Tutorials. [Online]. http://download.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

[132] Oracle, Sun Microsystems. Java. [Online]. http://www.java.com

[133] S. P. Overmyer, *Revolutionary vs. Evolutionary Rapid Prototyping: Balancing Software Productivity and HCI Design Concerns*. Fairfax, Virginia, USA: George Mason University.

[134] David L. Parnas, "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, pp. 1-9, 1976.

[135] J. B. Pawley, *Handbook of Biological Confocal Microscopy*. Springer, 2006.

[136] D. E. Perry, S. E. Sim, and S. Easterbrook, "Case studies for software engineers", in *29th Annual IEEE/NASA Software Engineering Workshop - Tutorial Notes*, 2005, pp. 96-159.

[137] Benjamin Pierce, *Types and Programming Languages*. The MIT Press, 2002.

[138] Python developers. Python Programming Language. [Online]. http://www.python.org

[139] Stefan Ram. Meaning of "Object-Oriented Programming". [Online]. http://www.purl.org/stefan_ram/pub/doc_kay_oop_en

[140] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture", *Computing Surveys*, vol. 9, no. 1, pp. 62-102, 1977.

[141] S. Redwine and T. Riddle, "Software technology maturation", in *Proceedings of the 8th International Conference on Software Engineering*, 1985, pp. 189-200.

[142] Trygve Reenskaug, "The Common Sense of Object-Oriented Programming", Department of Informatics, University of Oslo, Oslo, Norway, 2009.

[143] Trygve M. H. Reenskaug, "The original MVC reports", University of Oslo, Oslo, 1979.

[144] Trygve Reenskaug and James O. Coplien, "The DCI Architecture: A New Vision of Object-Oriented Programming", *Artima Developer*, Mar. 2009.

[145] Dirk Riehle, "Framework Design: A Role Modeling Approach", Zürich, Ph.D. Thesis 2000.

[146] D. Robson, "Object-oriented software systems", *Byte*, vol. 6, no. 8, pp. 74-86, 1981.

[147] C. Robson, *Real World Research*, 2nd ed. Blackwell, 2002.

[148] M. Rosson and S. R. Alpert, "The cognitive consequences of object-oriented design", *Human Computer Interaction*, vol. 5, no. 4, pp. 345-379, 1990.

[149] G. Rothermel and M. J. Harrold, "Analyzing Regression Test Selection Techniques", *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529-551, 1996.

[150] Winston Royce, "Managing the Development of Large Software Systems", in *Proceedings of IEEE WESCON*, 1970, pp. 1-9.

[151] Per Runeson and Martin Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical Software Engineering*, vol. 14, pp. 131-164, 2009.

[152] Frank Sauer. Metrics 1.3.6. [Online]. http://metrics.sourceforge.net/

[153] Ken Schwaber, *Agile Project Management with Scrum*. Microsoft Press, 2004.

[154] Ken Schwaber, "SCRUM Development Process", in *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1995, pp. 117-134.

[155] Ken Schwaber and Jeff Sutherland, *Scrum. The Official Guide*. Scrum.org, 2010.

[156] Prashant Sharma. TechPluto. [Online]. http://www.techpluto.com/web-20-services/

[157] SharpCrafters, "Producing High-Quality Software with Aspect-Oriented Programming", SharpCrafters, Technical White Paper 2011.

[158] Mary Shaw, "What makes good research in software engineering?", *International Journal on Software Tools for Technology Transfer*, vol. 2002, no. 4, June 2002.

[159] James Shore and Shane Warden, *The Art of Agile Development*. Sebastopol, CA: O'Reilly Media, 2008.

[160] Sinan Si Alhir, *Guide to applying the UML*. Springer, 2002.

[161] Paul Smith. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/wiki/File:Waterfall_model.svg

[162] C. Snook and M. Butler, "UML-B: Formal modelling and design aided by UML", *ACM Transactions on Software Engineering and Methodology*, vol. 15, no. 1, pp. 92-122, 2006.

[163] Joel Spolsky. Joel on Software. [Online]. http://www.joelonsoftware.com/articles/AardvarkSpec.html

[164] StartUML development team. StarUML - The Open Source UML/MDA Platform. [Online]. http://staruml.sourceforge.net/en/about.php

[165] D. Steffen, "The Purpose of System Testing", *Information Management Magazine*, no. July/August, 2010.

[166] Mark Stefik and Daniel G. Bobrow, "Object-Oriented Programming: Themes and Variations", *AI Magazine*, vol. 6, no. 4, pp. 40-62, 1985.

[167] Sun Microsystems. Java. [Online]. http://java.sun.com/docs/white/langenv/Intro.doc2.html

[168] Jeff Sutherland. Scrum Log Jeff Sutherland. [Online]. http://scrum.jeffsutherland.com/2003/02/scrum-keep-team-size-under-7.html

[169] The Institute of Electrical and Electronics Engineers, *IEEE Standard Computer Dictionary*. IEEE, 1991.

[170] Dave Thomas. (2011, Nov.) C2 Wiki. [Online]. http://c2.com/cgi/wiki?AspectsAndDynamicLanguages

[171] TIOBE. TOPBE Software: Tiobe Index. [Online]. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[172] Unforgiven.pl. ReThink - Challenge Yourself. [Online]. http://www.rethink.pl

[173] Unforgiven.pl. (2010, Mar.) ReThink - Trac. [Online]. http://trac.unforgiven.pl/rethink

[174] William Wake. (1998) Growing Frameworks in Java. [Online]. http://xp123.com/wwake/fw/

[175] Dean Wampler. (2011, Nov.) Object Mentor. [Online]. http://www.objectmentor.com/resources/articles/AOP_in_Ruby.pdf

[176] Peter Wegner, "Concepts and paradigms of object-oriented programming", *ACM SIGPLAN OOPS Messenger*, vol. 1, no. 1, pp. 7-87, 1990.

[177] Don Wells. Extreme Programming: A gentle introduction. [Online]. http://www.extremeprogramming.org/values.html

[178] Don Wells. Extreme Programming: A gentle introduction. [Online]. http://www.extremeprogramming.org/rules.html

[179] Don Wells. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/wiki/File:XP-feedback.gif

[180] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences", in *Proceedings of 11th International Symposium on Software Reliability Engineering*, San Jose, 2000, pp. 110-121.

[181] Wikipedia contributors. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Software_testing&oldid=384823849

[182] Wikipedia contributors. Wikipedia, the Free Encyclopedia. [Online]. http://en.wikipedia.org/wiki/File:Scrum_process.svg

[183] Wikipedia contributors. Wikipedia, the free encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Scrum_(development)&oldid=383854649

[184] Wikipedia Contributors. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=ISO/IEC_9126&oldid=408552093

[185] Wikipedia Contributors. Wikipedia, the Free Encyclopedia. [Online]. http://en.wikipedia.org/wiki/File:Confocalprinciple_in_English.svg

[186] Wikipedia contributors. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Software_prototyping&oldid=394516037

[187] Wikipedia contributors. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Criticism_of_Java&oldid=428806901

[188] Wikipedia contributors. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Object-oriented_design&oldid=389294192

[189] Wikipedia contributors. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Special:Cite&page=Data,_context_and_interaction&id=461011833

[190] Wikipedia contributors. Wikipedia, the free encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Extreme_Programming&oldid=382700417

[191] Wikipedia contributors. Wikipedia, The Free Encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Inheritance_(object-oriented_programming)&oldid=391610265

[192] Wikipedia Contributors. Wikipedia, the Free Encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=394517510

[193] Wikipedia contributors. (2013, Feb.) Wikipedia, the free encyclopedia. [Online]. http://en.wikipedia.org/w/index.php?title=Feature-driven_development&oldid=540470545

[194] L. Williams, "Integrating pair programming into a software development process", in *Proceedings of the 14th Conference on Software Engineering Education and Training*, Charlotte, 2001, pp. 27-36.

[195] N. Wirth, "Program development by stepwise refinement", *Communications of the ACM*, vol. 14, no. 4, 1971.

[196] E. Yourdon, K. Whitehead, J. Thomman, K. Oppel, and P. Nevermann, *Mainstream Objects: An Analysis and Design Approach for Business*. Upper Saddle River, NJ: Yourdon Press, 1995.

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Division for Natural Sciences and Technology
- Department of Information Technologies

ISBN 978-952-12-2920-6
ISSN 1239-1883

Mikołaj Olszewski

Scaling up Stepwise Feature Introduction to Construction of Large Software Systems