UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Chair of Software Systems

Sven Heiberg

Methods for Improving Software Quality: a Case Study

Master's Thesis

Supervisor: Jüri Kiho

Author: ...."............" 2002

Supervisor:...................................."............" 2002

Head of the Chair: ...."............" 2002

Tartu 2002

TABLE OF CONTENTS

I. Introduction 4
1. Prologue 4
2. Clarification of terms 5
3. Setting the goal 9
II. Project I – TrueSign 1.0 12
1. Goal of the project 12
2. Design workflow in software development process 13
3. Design activities 14
4. Communicating the design 16
5. Design quality 17
6. Design itself 18
III. Project II – TrueSign 1.1 24
1. Goal of the project 24
2. Design workflow in software development process 24
3. Design activities 25
4. Communicating the design 27
5. Design quality 28
6. Design itself 29
IV. Projects III and IV – Crypto Library 35
1. Goal of the project 35
2. Design workflow in software development process 35
3. Design activities 36
4. Communicating the design 37
5. Design quality 38
6. Design itself 39
V. Comments on software development process in TrueSign projects 44
VI. Design activities 47
1. Introduction 47
2. Communicating the design 47
3. Code reviews and pair programming 48
4. Visual modeling 51
5. Unit testing 55
6. Refactoring 58
7. Test-first design method 60
8. Alternative design methods 63
VII. Conclusions and further research 65
VIII. Resümee 69
IX. References 72

I. INTRODUCTION

1. Prologue

Software engineering is a field with many challenging problems, most of which can be tracked down to one thing – software is soft, it is easy to modify the software. In a broader sense, the software development process itself can be considered as continous cycles of modifications, changes, improvements and extensions to the current software configuration. Since it is so simple to make modifications into an existing implementation it is also easy to break the quality of the implementation. In their everyday tasks software engineers must deal very carefully with the change – it is important to be able to change software operatively while at the same time still maintaining its quality.

From May 1999 to June 2001, the author was part of a team in Cybernetica [Cyber Web], responsible for implementing an application which made extensive use of digital signatures. From October 2000, the author was one of the two software architects in Cybernetica, responsible for designing an application programmer's interface (API) for digital signatures. This API was meant to manage signed data and it also supported time stamping and notarization of signatures. The implementation phase revealed that there were several deficiencies present in the API so in January 2001 the author decided to use this lesson to design a new API which would include only most important parts from the previous one. In July 2001, the author decided to try test-first design method for the construction of the API.

This thesis comprises the software design related experience gained during the work. The methods used and errors made during projects are analyzed, ways to solve the encountered problems are suggested.

The thesis should be of interest to software developers and people close to the field of software development. The author sees the average reader as a person who is or will be a developer in a project where several people are working together on the same software system. This thesis should give the readers some points to think about while working on their own projects.

The author's focus is not on cryptography API in this thesis. There exist plenty of libraries that implement various sets of cryptographic primitives. Those libraries ([OSSL], [cryptlib], [Crypto++]) are thoroughly tested and highly optimized. Also several attempts have been made to standardize an API ([PKCS11]). A good discussion about design of an API for cryptographical primitives is given in [Gut99].

2. Clarification of terms

Throughout the thesis the terms 'software design', 'architecture' and 'development process' will be used. In this sub-section the definitions of the terms are given.

The organizational element through which software development is managed is called **project.** [JBR98]

By **product**, artifacts created during the life of the project, such as models, source code, executables, and documentation, are meant. [JBR98]

Software development process (development process, process) is a definition of the complete set of activities needed to transform user requirements into a product. [JBR98] Software development process itself consists of several sub-processes such as gathering user requirements, project management or programming. Those sub-processes are called **workflows** in this thesis. Each

workflow consists of **activities** representing a unit of work that can be performed by a person having a certain role in the software development process.

The organization and structure of the software product can be considered from various abstraction levels:

On the first level, there is the system that takes some input from the user and produces output according to the input. Here the static and dynamic structure of the software is defined only by **user requirements**.

On the second level, there is **architecture of the software**. Philippe Kruchten ([Kru98]) sees architecture as something that remains when one cannot take away any more details from the software description and still understands the system and is able to explain how it works. In the following, the author relies upon the Rational Unified Process (RUP [Kru98]) definition of the software architecture:

Architecture encompasses significant decisions about

- the organization of a software system;
- the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaboration among those elements;
- the composition of these elements into progressively larger subsystems;
- the architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition.

For a better understanding of the following discussion it is stated that architecture consists of strategic decisions upon which the software to be created shall stand.

The strategy of the architecture is executed by the tactical decisions made before or during the programming. Those tactical decisions are usually thought of as software design and they describe the static structure and dynamic behavior of the system using the appropriate notation and level of abstraction.

On the last level, there is the **source code** (code) of the software. Here all the algorithms and their implementations are fixed, every interface has an implementation and, with the help of appropriate tools, this description can be turned into executable machine code.

It could be noted that depending on the size of the software, several (starting from zero) abstraction levels between the architecture of the software and the source code can be defined.

Design of the software is a property of software that describes its static and dynamic structure.

Software design as a thing in itself is not very useful, it must be made graspable for the developers and here the previously described abstraction levels are useful from two viewpoints:

- describing the wished design of new software without considering unimportant implementation details;

- describing the general design of existing software to make it easier to understand its structure and behavior without considering unimportant implementation details.

Model is an abstraction of a system, specifying the modeled system from a certain viewpoint and at a certain level of abstraction [JBR98].

Software design can be represented as a set of models – **design models** – that describe how user requirements and strategic decisions of the architecture are going to be fulfilled by software. It must be possible to create a product fulfilling user requirements according to design models.

Design models are means for communication between software developers. A model that fulfills user requirements but cannot be properly understood by most of the developers cannot be used to build the needed software.

Software design method (design method) as defined by David Budgen [Bud93] is a structured way of creating the software design that consists of the following main components:

- representation – one or more forms of notation that can be used to describe (or model) both the structure of the initial problem and that of the intended solution, using one or more viewpoints and differing levels of abstraction;

- process – description of the procedures to follow in developing the solution and the strategies to adopt in making choices;

- heuristics or clichés – guidelines on the ways in which the activities defined in the process part can be organized for specific classes of problem.

Software design activity is a process during which the level of abstraction is reduced, i.e. the strategical decisions on one level are turned into the tactical decisions on the next level. Whenever speaking of designing software, this activity is meant.

Software unit is a piece of source code that logically belongs together such as a class in Java or C++. One software unit can consist of several smaller software units.

3. Setting the goal

It was noted that several levels of abstraction could be defined between the architecture of the software and the real representation of software – the source code. The number of the levels depends on the complexity of the software. What for some application is called architecture might be far too detailed for another application where better understanding of the system would require more abstraction levels.

Similar issues arise when discussing the software development process. This process consists of several more specific processes – workflows – that all are very important in the overall process. Each of those workflows must also have clearly defined structure and dynamics. One must understand what to do and why – the success of software development process builds upon the success of its contents. In this thesis the author has the focus on the design workflow and, more precisely, on the design activities contained in this workflow.

The author identifies the communication as one of the most important parts of software development. Some techniques that enforce communication, such as pair programming, are examined in depth.

The author identifies the need for a method on all levels of software development. In the author's opinion if one part of the process is done without conscious knowing of what and why then, in the long run, this can cause severe problems. The author also discusses some design methods – and particularly test-first design method – that can be used for writing source code. The author feels

that although there has been a fair amount of research made on software development and software development methods, still enough insight has not been given into the process of producing the final design efficiently and with high quality.

All the author's statements made in the thesis are based on the author's own experience gained through working in four projects – TrueSign 1.0, TrueSign 1.1 and two experimental projects. Those projects will be referred to as TrueSign projects in this thesis.

The next three chapters will contain descriptions of the TrueSign projects. Each of them will consist of the following sections:

- Goal of the project – this section deals with the project's background. The goal of the project is also stated here.

- Design workflow in software development process – the relative position of design activities in overall software development process is described, the time amount that was spent for designing the software is given and the documents that existed before the design started are observed. Also, the mechanisms for measuring the project, if applicable, are discussed here.

- Design activities – in this section the design process on the developer level is discussed and the design methods used in the project are described.

- Communicating the design - here the focus is on the communication mechanisms used in the project. The author observes how the design decisions were documented and passed on from developer to developer.

- Design quality – in this section the quality assurance mechanisms such as testing used in the project are examined.

- Design itself – in this section the software design models are presented

Chapter V contains the author's notes on the software development process applied in the TrueSign projects. Chapter VI discusses problems related to design activities. Chapter VII contains the most important conclusions of this thesis, some topics for further research are outlined in this final chapter also.

II. PROJECT I – TRUESIGN 1.0

1. Goal of the project

Since its founding in 1997, Cybernetica [Cyber Web] has done active research in various information security related fields. The focus of Cybernetica's researchers has been on digital timestamping and notarization of electronic documents.

Until recently, validity of a digital signature on an electronic document could be successfully verified only if the certificate issued to the signer by some certification authority (CA) was valid. This property made use of public key infrastructure (PKI) unfeasible in many fields of economy, e.g. such as banking – certificates are usually valid up to three years, but validity of some signatures (e.g. in bookkeeping [Law384]) must be verifiable even ten years after their creation.

The goal of the TrueSign 1.0 project was to demonstrate the technology which would enable verifying that the certificate was valid at the time of signing, even if the verification took place after the certificate has lost its validity.

The architecture of the software system created during TrueSign 1.0 project was complex. This thesis only focuses on the subsystem of cryptographic primitives. The main goal of this subsystem in TrueSign 1.0 was to provide upper layers with facilities for signature generation, signature verification and message digesting. Supported mechanisms were the SHA-1 and MD5 message digesting algorithms and the RSA signature scheme.

Cryptography and digital signatures are explained in [Sti95]. Information about digital timestamping can be found in [Lip99]. Notarization with timestamping and

long-term validation of digital signatures are described in [ABRW01] and [Roo99]. TrueSign specific web pages can be found from [TrueSign Web].

2. Design workflow in software development process

Previous experiences with projects where no defined software development process was used, lead the development team of Cybernetica to evaluate the Rational Unified Process (RUP, [RUP Web] [Kru98]) framework while developing TrueSign 1.0.

The whole project was divided into four standard RUP phases – inception, elaboration, construction and transition phase.

The inception phase included 4 man-days of use case gathering; other activities were not so directly related to software design. Inception phase lasted 44 man-days altogether.

During elaboration phase, use case analysis continued and the design of most of the whole system (two complex servers and a library for client software) took place. Directly design related activities lasted 32 man-days and the result of this phase was a model in Unified Modeling Language (UML, [UML Web]) where all the components of the software system were identified and decomposed into ca. 150 classes with ca. 750 methods. At the end of the construction phase the system consisted of ca. 200 classes and ca. 2200 methods (including constructors and destructors).

Two papers accompanied this UML model. One of them described the theoretical background of the system, the other (TrueSign whitepaper) attempted to describe the most important data structures, protocols and their compliance with standards such as [X509] and [CMS]. UML model and the TrueSign

whitepaper changed rapidly during the construction phase; the theoretical paper was stable and remained unchanged.

The construction phase lasted 271 man-days. During those days seven developers implemented the components designed in the previous phases. 6 man-days were spent on coding the subsystem of cryptographic primitives. At the end of the construction phase, every developer was required to write an essay which would express his opinion about the project from any aspect he might find interesting or useful. In addition to the author's own experience, these essays are another source for the data analyzed here.

3. Design activities

Design activities were spread over two phases in TrueSign 1.0. The components of the system were identified during the elaboration phase. Instead of giving plain component and interface definitions, identified components were decomposed into smaller units by one software architect and the result of this phase was a class diagram of the whole system. For every group of 1 – 3 related classes, the architect gave an estimation for the time of development. Those estimations were used to plan the whole construction phase.

The documentation of a class was usually quite sparse, consisting only of one or two sentences. The purpose of a subsystem and the interactions of its classes were usually not specified. This forced the developers to guess what functionality a class or a group of classes must offer to the rest of the system. This attempt usually resulted in a phone call to the architect who then explained the meaning of the class and reasons behind some of the design decisions.

When a developer found that enough information was gathered from the model with the help of the architect, coding of the class was started. When all the methods of the class were finished then sometimes tests were written.

In short, the design process in the construction phase for the developer included the following steps:

1. Reading the model and understanding at least some part of the task.

2. Contacting the architect and verifying that the model has been understood correctly.

3. Implementing the part of the model where both the architect and the developer have agreed on the structure of the implementation.

4. In case of any unimplemented classes or methods in the task, repeating from the first step.

5. If time allows, writing some tests to prove that the code works; correcting the flaws.

6. Updating the model in case of any major changes made in the system structure.

In the middle of the project, after realizing that the deadline cannot be met, the project management decided to gather data about the activities a developer had to perform during a workday. Every developer was told to keep a logfile where every activity, such as phone call from other developer or going to lunch, would be written down. The idea behind the plan was to discover disturbing factors and to eliminate those factors in the following projects.

4. Communicating the design

During the TrueSign 1.0 project the Cybernetica team tried to use visual modeling with Rational Software development tools. Several developers were not happy with the tools and visual modeling in general. The main reasons were that the tool support was insufficient and some developers found it easier to think in terms of programming language rather than in terms of UML. The source code and visual models had to be kept in sync by hand. Unfortunately it cannot be told how much time the developers spent updating models as the logging process that could have gathered the data was started in the middle of the project where most of the developers did not update the model any more because of the time pressure.

Verbal communication had an important role in TrueSign 1.0. One could have distinguished between three types of verbal communication in the project listed below.

The driving force behind the first type of communication was the lack of documentation in models. When somebody started to implement a new package first the model was studied and then usually the architect was contacted to make sure that the task had been understood correctly. There was only one person, located 185 km away from the rest of the development team, who knew the purpose of all subsystems and his time was very valuable. Also the internals of some subsystems were only known to those developers who had authored them.

Second type of communication occurred when somebody tried to use any of the existing components. Then usually the author of the component was asked to specify more precisely what could be done with the component and how. The driving force behind this type of communication was developer's wish to go on

with the personal task as fast as possible and also the lack of documentation in the model and the source code.

There was also a strong communication line between the developers and the project management. Developers were required to write daily reports to the project management. Those reports would consist of the identification of the developer and the task at hand with description of the work done the day before, plans for the present and the next day, plus an overall estimation about the task's duration. Also, the daily logging, introduced in the middle of the project, strengthened this communication line.

5. Design quality

In TrueSign 1.0, almost no explicit steps were taken to assure the quality of the product. Developers were not required to write tests. The overall agreement on what task the system must accomplish was maintained by personal communication between the developers and the software architect.

Integration of different subsystems started on the last week of the construction phase. In the beginning it was scheduled that five developers work for two days and test the most important use cases. The amount of defects discovered that way was huge, including also some conceptual problems such as misunderstandings about data formats, etc. At the end, the time that was spent on integrating subsystems was 25 man-days, most of which was spent doing over-time.

However, there existed one set of design quality and communication related rules – code style rules. As the TrueSign 1.0 team consisted of seven developers, everybody of whom having their own personal coding style preferences, it was decided to define certain rules to make it easier for everybody to understand each

other's code. The rules were not followed strictly and the developer who had written the code could usually be identified by the coding style.

6. Design itself

This thesis focuses on the design of the subsystem of cryptographic primitives of TrueSign 1.0. The reason is that this subsystem is the common theme in all four projects.

Instead of writing the crypto subsystem from scratch, Cybernetica chose to fit an existing crypto engine for its purposes. The subsystem described here is basically a wrapper around OpenSSL [OSSL] engine. OpenSSL is an open source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. The reasons for picking OpenSSL were the following:

- OpenSSL is an open source toolkit. The developers themselves can fix any defects in the engine. If the documentation is sparse (which is exactly the case with OpenSSL), one can always consult the source code to find out the real semantics of a function.

- Most of developers were already familiar with OpenSSL. Cybernetica had used it in other projects as well and any newcomers were introduced to it sooner or later.

- OpenSSL is used extensively in various (non-) commercial software products (more than 70 references); it is thoroughly tested and ported to most known platforms. Low-level operations are highly optimized.

Throughout the thesis UML ([UML Web]) notation is used to describe the software design in the figures.

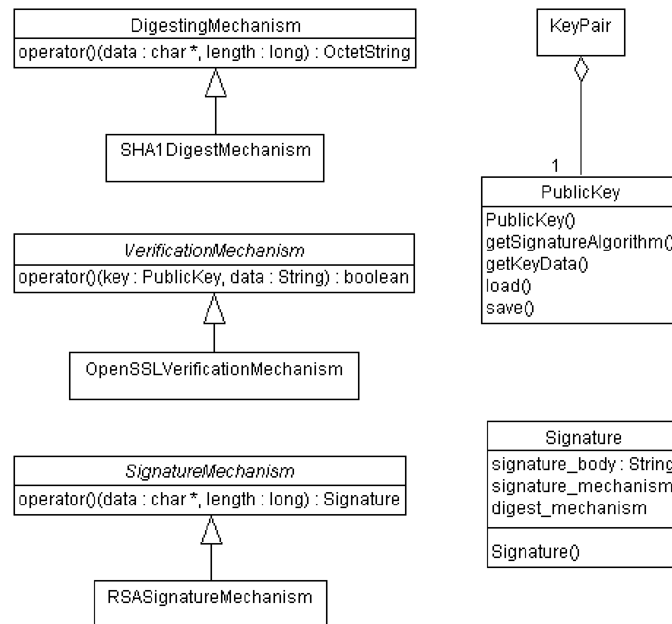

Figure 1: Initial description of a crypto subsystem

Figure 1 describes the design of cryptographic primitives subsystem as it was handed out to the developers. The API evolved during the development as its requirements became clearer.

During the development it was not sure which signature methods are going to be supported and which not. To make an addition of new methods easy, two abstract interfaces were defined for public keys (*PublicKey*) and public-private key pairs (*KeyPair*) (Figure 2).

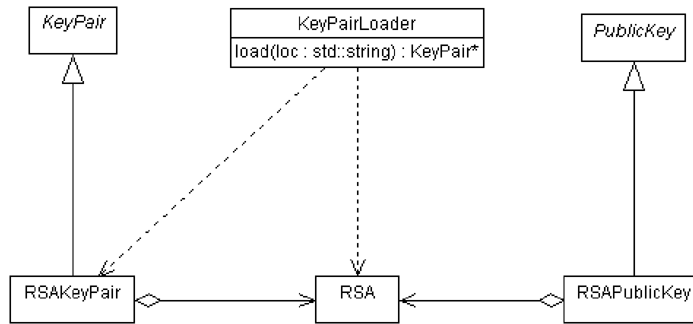

Figure 2: KeyPair and PublicKey

Before every signing operation, a private key had to be loaded. To prevent that every client program would have to know about every possible private key type, a factory class *KeyPairLoader* was introduced. The program would have created an instance of *KeyPairLoader* class and called its *load* method with argument showing the location of the key pair file. User would have gotten an object confirming to type *KeyPair* in return. This object could have been used to initialize a signing process.

Similar solution, a modification of the Factory Method design pattern ([GHJV95]), existed also for public keys. It was in *Certificate* class, outside of the scope of the cryptographic subsystem. The solution was sufficient for the TrueSign 1.0 project, as the only way a public-key could have entered the system was through a certificate.

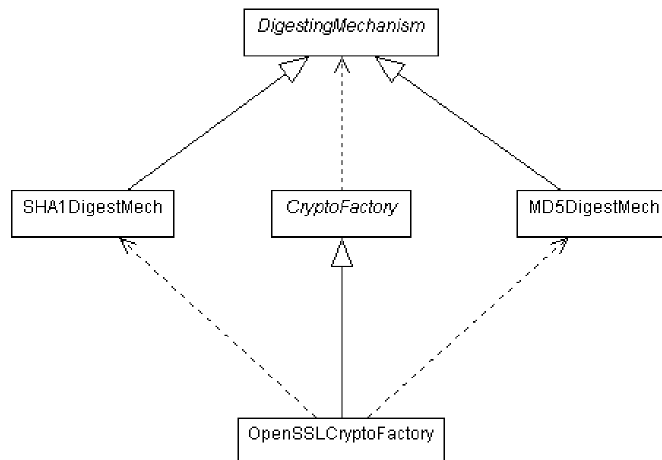

Figure 3: Digesting and factory class

The actual implementation of message digesting (Figure 3) consisted of an abstract interface *DigestingMechanism*. All concrete mechanisms inherited from the interface. To save client programs from having to know every descendant of *DigestingMechanism* class and to allow addition of another crypto engine in the future, an abstract factory class *CryptoFactory* was introduced. Descendants of the *CryptoFactory* class contained the logic that distinguishes between different mechanisms and knows how to create them with the current engine. The instance of this concrete class was accessible to other components via global variable.

A particular design problem can be pointed out here. The *KeyPairLoader* class is also specific to the crypto-engine used (in Figure 2, one can see how it depends on the *RSA* which is OpenSSL specific structure). It would have been better to make a simple structural change in the code and move the public key loading method from *KeyPairLoader* class to *CryptoFactory* class to reduce the possibility of errors that occur if user initializes *SignatureMechanism* instance created by one engine with a *KeyPair* instance created by another engine.

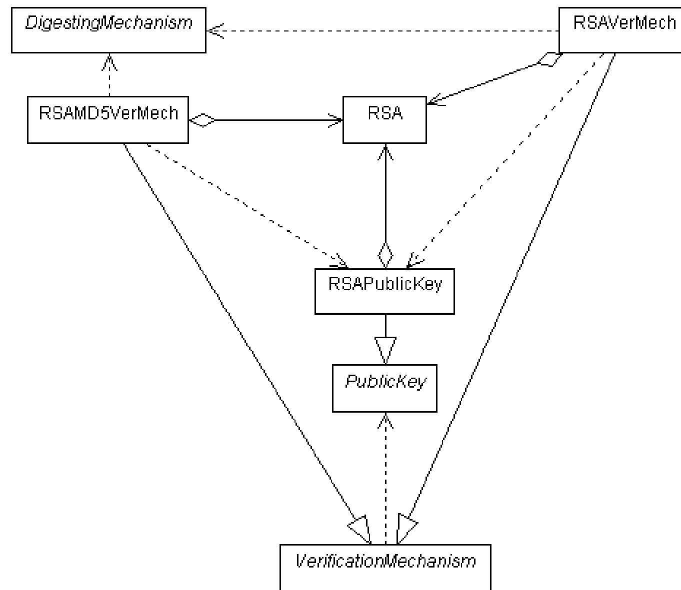

Figure 4: Verification mechanisms

Implementation of verification primitives can be seen in Figure 4. *VerificationMechanism* was an abstract interface used by clients. All the verification mechanisms were created with the help of global instance of *CryptoFactory*. To be usable, an instance of type *VerificationMechanism* had to be initialized with an object confirming to type *PublicKey*, which in case of the RSA based mechanisms had be of type *RSAPublicKey*.

The problem with the design is that there are two subclasses which implement RSA verification. Those two differ only in the digesting mechanism used by them. As the code only refers to *DigestingMechanism* interface, those two classes again could be brought together with a simple structural change in the design.

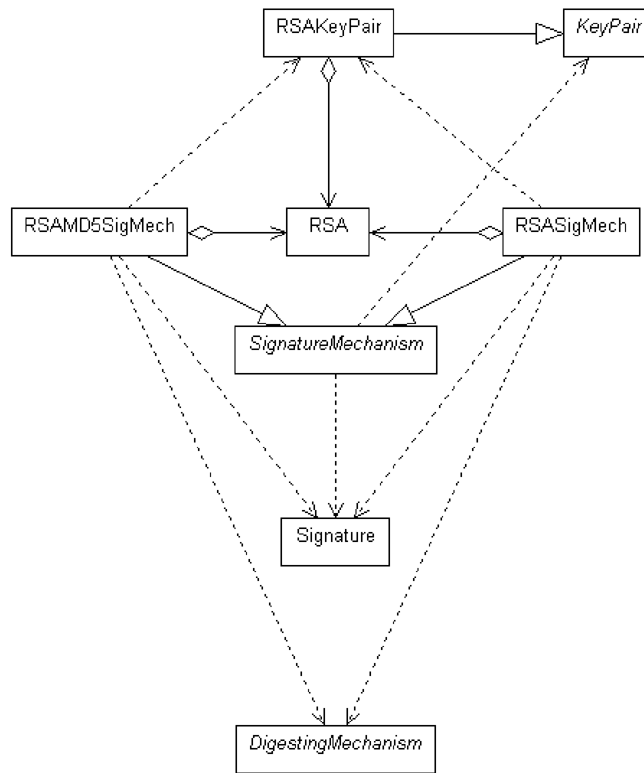

Figure 5: Signature mechanisms

The *SigantureMechanism* implementation presented in Figure 5 was dual to *VerificationMechanism* implementation (Figure 4).

III. PROJECT II – TRUESIGN 1.1

1. Goal of the project

The results of TrueSign 1.0 project were two servers and one client, which were applicable as technology demo. The goal of the follow-up project TrueSign 1.1 was to turn the TrueSign into a marketable product. Overall goal included two sub-goals. Firstly, the research team redesigned the protocols used in TrueSign 1.0. Secondly, an API, which could be used by any software vendor who would like to add digital signature capabilities to programs, was to be designed. Merely the API design part of the process is analyzed in this thesis.

2. Design workflow in software development process

The analysis of the data gathered in TrueSign 1.0 project had given useful information which made task planning and estimating during the TrueSign 1.1 easier compared to previous projects; it was decided that the team follows the RUP guidelines in this project, too. There were some modifications made to RUP recommendations, though. Inception phase was discarded. The management decided that there was no need for the inception phase as all use cases, risks and goals from the TrueSign 1.0 would still apply.

In the elaboration phase two architects were given a task to design the API. Those architects spent about a week analyzing the suggestions made by co-developers, analyzing APIs made by other vendors, reading relevant standards, etc. After that, a three-week period was dedicated to design activities. Two architects were working together as a pair behind one computer. Most of the design decisions were made together. Architects tried to avoid the situation where the API would evolve during the development in inconsistent ways as had

happened during TrueSign 1.0 so during this three-week period, a full three-layered API was designed. The layers were the following:

- A crypto subsystem that contained cryptographic primitives.
- A low-level client subsystem that contained relevant data structures and standardized signing procedures.
- A high-level client subsystem that was meant to be used by a programmer who would not know too much about digital signatures and would not want to customize their use.

The results were presented as an UML model, consisting mainly of class diagrams with ca. 250 classes and ca. 1150 methods. At the end of the construction phase the system consisted of ca. 300 classes and ca. 2600 methods (with constructors and destructors).

After finishing the design and the documentation one of the architects was sent to a vacation for a month. The other architect followed him shortly and took a week's vacation. There were no architects present in the development team for the initialization of the construction phase.

This time, nine developers were working in the construction phase and there was altogether 650 man-days of work to do. 90 man-days were spent in coding the crypto subsystem. The construction phase was planned to have three iterations. Altogether, it took 145 days to finish the phase.

3. Design activities

As in TrueSign 1.0, here again class diagrams with most of the classes and methods specified were handed out to developers as the input for their work.

This time the developers' task was harder though. The architects had invested lots of brainpower into generating beautiful constructs to solve various design issues. For example, the architects had made a decision to eliminate the presence of the low-level crypto-engine from public interface because they had to support several engines. This decision resulted in yet another layer of indirection, which was not defined too clearly. In the beginning of the construction phase, the architects were also few weeks absent from the development process and thus several general design issues were understood and solved in inconsistent ways. This lengthened the learning curve for the developers who joined later in the process.

Another slight change to the design process on the developer level was that this time testing was made mandatory. After creating a software unit, a developer would also write tests for it and make sure that those tests would pass. The rule again was not followed very strictly.

The process can be outlined in the following way:

1. Reading the model with the documents included and understanding at least some part of the task.

2. Contacting the architect and verifying that one has understood the model correctly.

3. Implementing the part of the model which was understood.

4. Repeating from the first step, in case of any unimplemented classes or methods in the task.

5. Writing some tests to prove that the code works, correcting any defects.

4. Communicating the design

TrueSign 1.1 used the same means for design communication as TrueSign 1.0 – visual modeling, documentation and storytelling.

The developers of TrueSign 1.0 project found it inconvenient to keep the source code and the model in sync without real support from the tools. In TrueSign 1.1 it was decided to use source code as the basis for the truth. Reverse-engineering facilities in visual modeling tool were used to update model from the code and not the other way round. This also proved to be quite a tedious task, but this time only one developer had to spend time on it.

Lack of documentation was one of the favorite discussion topics during the development of TrueSign 1.0. This time plenty of the architects' time was spent documenting the UML model for other developers. Every class and method in the class had up to five-sentence comment describing its purpose. There were also code examples to explain some parts of the design and documents to describe some general solutions in more detail.

As there was more emphasis on documentation, the need for storytelling was not so big anymore. Developers also had two architects to turn to this time so the load was balanced and both architects also had time to work with the code. In fact, several critical parts of the API, such as standardized verification primitives, were written or re-written by architects.

Those two architects were working in two different cities so there was no face-to-face contact between them during the construction phase. In addition, the development team was divided between two cities.

The communication line between project management and developers was again supported by daily reports. In TrueSign 1.1 no logging of developers' activities was done.

5. Design quality

During TrueSign 1.1 developers understood that there is a need to ensure product quality in some way. The problems started when some units' functionality, upon which the others would rely, were changed. To make sure that adding new code will not break existing functionality, an explicit rule that made testing mandatory was introduced to the project.

While testing was forced on by the will of developers and project rules, code reviews happened spontaneously. Those code reviews had purposes such as correcting a defect in code written by someone else or trying to understand how to use a part of the API in one's own code. Both types of reviews helped to spread the knowledge about design decisions made during the development of one unit.

Architects also read the code of more important modules to make sure that their guidelines were followed. If the guidelines were not followed then the discussion with the author of the code indicated whether it was a defect to be repaired or a design decision that was forced upon by the reality and was overseen by architects who did not try most of their ideas out in code.

Modification of units written by other developers was possible because of one implicit rule used in programming process – there was no code ownership. If a piece of a source code was released to the repository then afterwards anybody was free to improve it. Usually it was done hand in hand with the actual author of

the code, who might have had some serious thoughts behind the somewhat awkward design.

Code style rules were used in TrueSign 1.1. Unfortunately the code style rules were not followed too strictly.

Evidently the rule of no code ownership is not useful when every developer writes code according to his or her own standards. When two developers modify the same code, two styles get mixed up and the result can be misleading in the worst case. The problem with some code style rules is that the tools used by the developers to modify the source code should support them also. If the rules for line wrapping differ between programmers and various editors, then it is hard to get all the developers to wrap their code the same way.

6. Design itself

In TrueSign 1.1, only two-thirds of the functionality foreseen by architects in the visual models was actually implemented. Further on, the design of hash mechanisms, signature and verification mechanisms and keystores will be described.

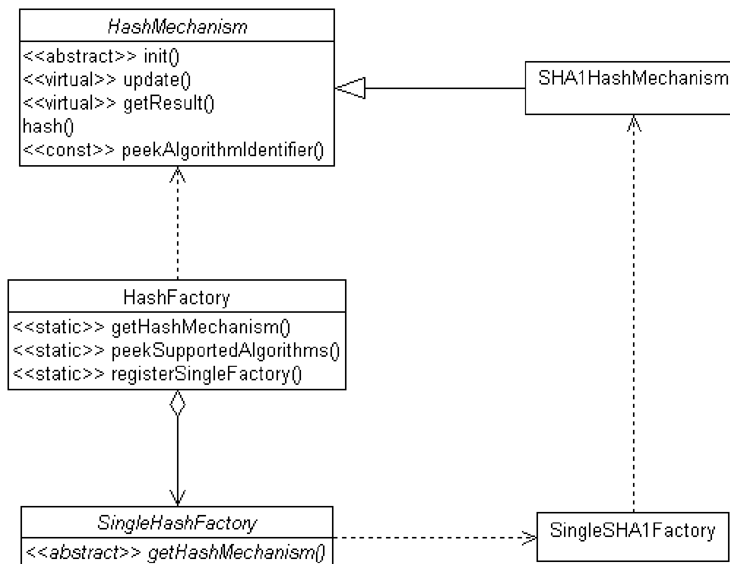

Figure 6: Hash mechanisms

The root of the hashing API (Figure 6) was the class *HashMechanism*. This class defined abstract interface for all the concrete hash mechanisms such as *SHA1HashMechanism* and the others. Only the abstract *HashMechanism* belonged to the public API. All concrete mechanisms were left into the implementation part and there was no possibility for the user of the API to instantiate them directly. Instead there was a single point of creation for all of them – *HashFactory*.

It was possible for the users to add their own hash functions to the library. For every concrete hash mechanism implementation, there was a special factory class in the system that knew how to create the concrete implementation. This factory class inherited its interface from *SingleHashFactory*. This interface was known to the *HashFactory* and could be used to register new hash mechanisms at run-time. A similar registration mechanism existed for the verification and signature mechanisms and for the key stores also.

During the implementation phase developers showed creativity and *SingleHashFactory* subclasses were rendered to engine specific factories. Their *getHashMechanism* methods took special algorithm identifiers as arguments and chose between several mechanisms implemented by a concrete engine. For example *OpenSSLSingleHashFactory*, inherited from the *SingleHashFactory*, knew how to create parameterized instances of *OpenSSLHashMechanism* class, which was a subclass of *HashMechanism*. Similar sub-factory was also created for other crypto engines such as Microsoft CAPI [MSCAPI].

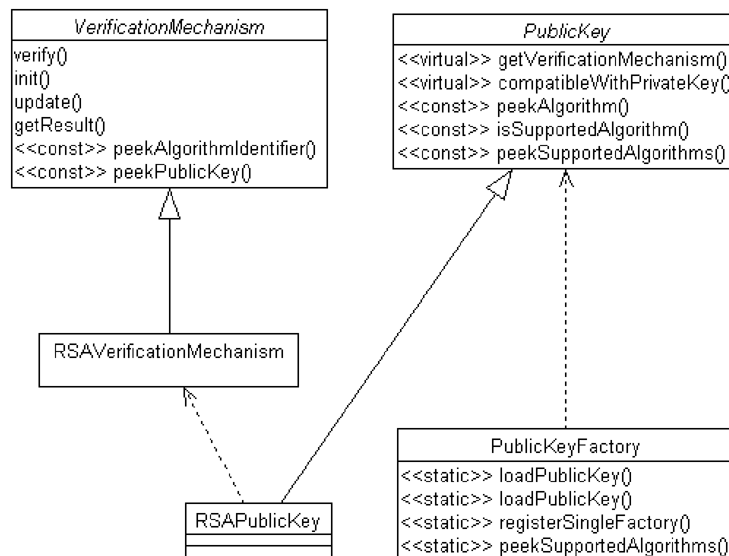

Figure 7: Verification and public keys

Figure 7 describes the verification and public key management in TrueSign 1.1. *VerificationMechanism* was an abstract interface for all concrete verification mechanisms. Concrete mechanisms belonged to the API implementation and were not directly visible to the user. User could create an instance of *VerificationMechanism* only if presented with an instance of *PublicKey* class.

The solution differs from the solution used in other crypto APIs [Crypto++]. It is logical as there can be no signature verification done without a public key. *PublicKey* is an interface for various types of public keys such as *RSAPublicKey*. Instances of effective subclasses of *PublicKey* interface only create verification mechanisms of certain type. An object of type *RSAPublicKey* will be able to create only objects of type *RSA VerificationMechanism*.

Also, instances of *PublicKey* could not be created directly by the API user. To extract public key from a sequence of bytes, one had to use *PublicKeyFactory* which was built similarly to the other factories in the API.

Here again, the actual implementation slightly differs from the ideas of architects. *OpenSSLSinglePublicKeyFactory* is capable of producing *OpenSSLPublicKey* instances which represents RSA public key. *SinglePublicKeyFactory* makes difference between crypto engines and not the types of public keys.

Implementation of signing API (*SigningMechanism*, *PrivateKey*) was dual to the implementation of verification API (*VerificationMechanism*, *PublicKey*). The only difference was in creation of *PrivateKey* instances. This paper focuses on storing private keys and creation of objects of type *PrivateKey*.

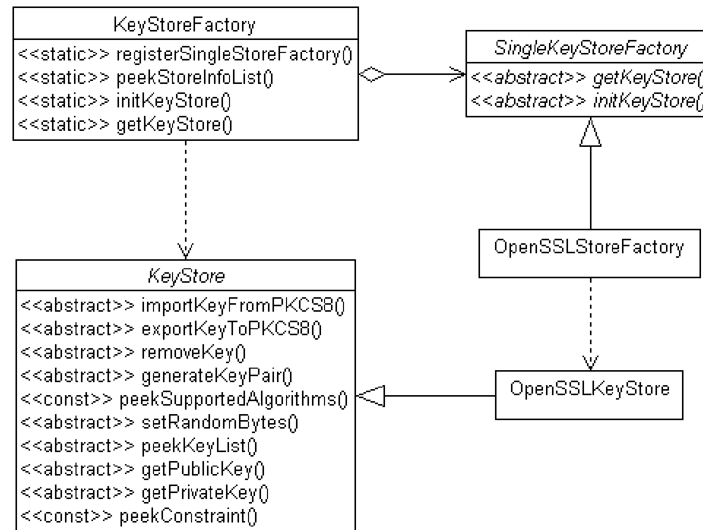

Figure 8: Key stores

Key store is a place where private and public keys can be kept in. A key store can be a smart card accessible via PKCS #11 [PKCS11] interface, or encrypted file in PKCS #12 [PKCS12] format. The cryptographic service providers in the Microsoft CAPI [MSCAPI] can also be seen as one type of key store. There exist lots of interfaces for managing different types of key stores. For TrueSign purposes, the architects tried to extract and unify some of the needed functionality, which was present in all of the relevant interfaces known to them.

The functionality of a key store consisted of adding keys to the key store, using and removing keys from the key store and generating new keys to be kept in there. This functionality was declared by *KeyStore* interface in TrueSign API (Figure 8). Subclasses of *KeyStore* implemented the functionality with respect to various storing methods as described previously. For example, *OpenSSLKeyStore* implemented storage in PKCS #12 format files.

As there can be many types of key stores, TrueSign API also contained *KeyStoreFactory* that could be used to register known types of key stores and use existing key stores that are of a certain type. It was also possible to generate new key stores (such as PKCS #12 files).

This design is quite generic and also not easy to implement. It is the youngest part of TrueSign API and thus not as stable as e.g. hashing API. There are plenty of questions that rise with the different natures (hardware vs. software based) of various key stores. The differences in design of PKCS #12 and Microsoft CAPI based stores caused quite a lot of trouble for the development team.

There are problems in the other parts of the API as well. In several places the developers have changed the nature of the solution slightly (moving from a mechanism-based inheritance to engine-based inheritance) without changing the existing framework (e.g. class names). Although those changes indicate a design problem – the proposed design is not as flexible as it could be – the solution is still halfway there and needs refinement.

In the signing and verification API there were parallel inheritance hierarchies such as *PublicKey* and *VerificationMechanism* where *PublicKey* was only useful for storing data about a *VerificationMechanism*. Those hierarchies could be merged into one hierarchy of *VerificationKey* and the same change could be made in the signing API as well, thus making it easier to add new mechanisms to the system.

IV. PROJECTS III AND IV – CRYPTO LIBRARY

1. Goal of the project

During and after the TrueSign projects, the author of this thesis tried out some ideas that were not applied in the projects because of the time pressure and the relative novelty of those ideas for the development team. This work consisted of two experiments, which are discussed together here because of their relatively small scope.

The goal of the first experiment was to produce an object-oriented API of cryptographic primitives. It was to be based on the TrueSign 1.1 API with some known errors fixed.

In the second experiment the author designed the same API, now using test-first design method. The goal of this experiment was to find out whether it would be beneficial to apply the method in real projects also.

2. Design workflow in software development process

First of the experiments lasted two months and two people were involved in it. One of them – the author of this thesis – was working out the design and the other, a more experienced person, was reviewing the design and making suggestions for improvements. This experiment was purely design specific and included almost no project management overhead. The only thing produced during this experiment was the design model of the API in UML.

The second experiment lasted one month. Mostly, only the author was involved, but for two weeks, another developer participated. Here the focus was on coding

and design, the project management overhead being minimal. This time test-first design method was applied and the main result of the experiment was a source code of the API with extensive set of tests.

3. Design activities

There is not much to learn from the design process of the first experiment. It included one developer doing work in a way that is hard to describe. The results of his unguided thinking were sent to another developer who reviewed the model and applied some refactorings (see section VI.6) to the design. To be exact, no defined design method and development process was used during this experiment.

The design strategy in the second experiment was as follows:

1. Designers agreed upon the task which was going to be solved.

2. A test was written to prove that the functionality that was not implemented yet worked as intended.

3. All tests were compiled. At this point the test added in Step 2 did not compile.

4. Interface for functionality was created, empty implementations were added.

5. All tests were run. At this point, the test added in Step 2 failed. If it did not fail, then the designers tried to find out why and then created test that would fail.

6. The simplest possible code was written to satisfy the test. Sometimes it was as easy as returning the right answer for the test case.

7. The code written so far was reviewed to spot any duplication, too long methods, etc. If something was detected, attempts were made to change it without breaking the tests.

8. If the designers could think of more tests then the whole process started again from Step 2.

4. Communicating the design

One of the persons participating in the first experiment was in Helsinki, Finland and the author of this thesis was in Tartu, Estonia. Those people used e-mail and UML models to communicate. The model was modified only by the author, the partner made suggestions that were possibly implemented. No code was written to prove the superiority of any design decision over another possibility.

Design patterns as in [GHJV95] were used as communication tool during the first experiment. Also, the design that was created was pattern oriented. Several problems from the design field were taken and solved with help of one or two quite complex design patterns.

Design pattern is a solution to a problem in context. Those solutions have evolved in the course of time; one solution is considered to be a pattern if applied to more than two software products. An important property of the design patterns is that they can be used as communication tools between developers. Implementation of a Visitor pattern [GHJV95], for example, is not an easy one and to describe it with just a few words is a hard task. If two developers both

know the Visitor pattern then it is sufficient enough to mention the usage of the pattern and both will know what is being talked about.

In the second experiment, visual modeling was abandoned. The main ideas already existed in models drawn during previous projects; it was now necessary to implement them properly. Whereas the first experiment used conversational design, pair programming technique was applied in the second experiment.

Pair programming is a way of working: two programmers have the same task and they solve it together behind one computer using the same monitor and the same keyboard. The technique is discussed in section VI.3.

Pair programming was done hand in hand with design method where the tests exist before the functionality that satisfies the tests. The result was an extensive set of tests which communicated the functionality and usage of the code.

5. Design quality

No explicit measures to guarantee the quality of the design were taken in the first experiment. The design just happened and it was not known whether it was any good or not – there was no possibility to execute the design.

In the second experiment, measures were taken to assure the code quality. A testing framework was created which made it possible to automatically detect whether added functionality had broken some existing functionality or not. Also, a technique called refactoring (section VI.6) was applied to keep the design evolving.

6. Design itself

Design made in the first experiment included public-key and secret-key cryptography, key generation and storage. Here a part of the solution offering hash functions to applications is described.

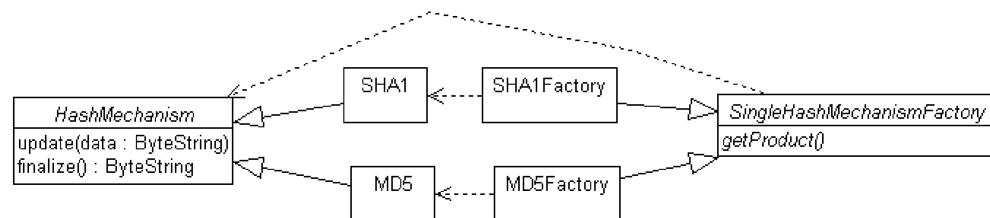

Figure 9: Design of hash functions

ByteString was a class that holds the buffer of bytes with the length of the buffer in it, thus replacing all the calls to *method (const unsigned char*, size_t)* with calls to *method(const ByteString&)*.

SHA1 and *MD5* were classes that implement specific hash functions. They had to implement *HashMechanism* interface (Figure 9). The latter had two methods – *update* and *finalize*. With the *update* method, a hash calculation gets data to process. Several subsequent calls to the *update* give the same result, as would have given one call with concatenated *ByteStrings*. This was necessary to support the hashing of very big amounts of data that become available in chunks.

Concrete hash mechanisms could not be created directly by the user of the API. Every hash mechanism had a corresponding factory class that implemented the *SingleHashMechanismFactory* interface. This interface was used to register factories of concrete hash mechanisms at one *HashFactory*, which was a single point of creation for the instances of *HashMechanism*. This instance of Abstract Factory pattern [GHJV95] is described in Figure 10:

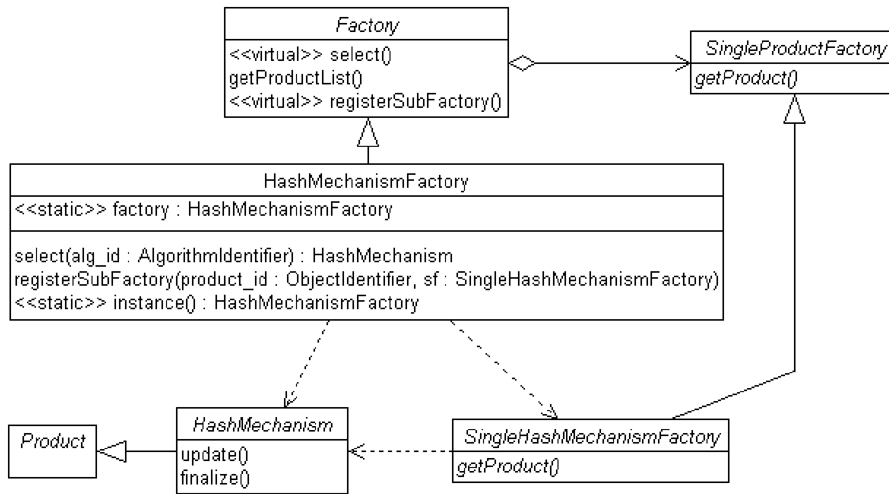

Figure 10: Abstract Factory pattern implementation

HashMechanismFactory was inherited from an abstract class *Factory*. The latter defined how to store *SingleProductFactory* instances and how to create *Product* instances with their help. To guarantee polymorphism, *HashMechanism* had to be inherited from *Product* interface and *SingleHashMechanismFactory* from *SingleProductFactory* interface. Note that *HashMechanismFactory* was necessary for only two things: it implemented Singleton pattern [GHJV95] and made sure that only *SingleHashMechanismFactory* instances got registered by it. All the logic was contained in *Factory* class and could thus be reused by other classes such as the signature and verification mechanisms.

HashFactory registered *SingleHashMechanismFactory* instances with *ObjectIdentifier* instances. *ObjectIdentifier* was the class that represented ASN.1 type OBJECT IDENTIFIER [ASN.1]. It was up to the user to register hash functions with standardized identifiers.

Retrieval of a hash function followed with the help of an instance of *AlgorithmIdentifier* class. *AlgorithmIdentifier* represented the standardized ASN.1 type AlgorithmIdentifier.

ASN.1 AlgorithmIdentifier consists of an object identifier and optional parameters that are coded as a sequence of bytes ([X509]). Object identifier represents an algorithm; the parameters contain additional information and are used by some algorithms. No hash function known to date currently uses them. The use of AlgorithmIdentifier structure was necessary in the project because most supported standards ([CMS], [X509]) used this structure to identify the cryptographic mechanisms used.

The design developed in the second experiment started off from the architectural ideas of the previous projects, which were then implemented with test-first design method. Here again the solution for the hashing API is described (Figure 11).

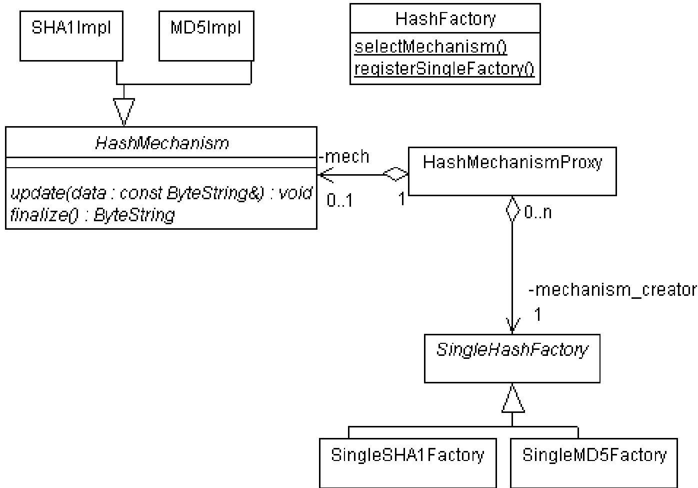

Figure 11: Design of hash functions

SHA1Impl and *MD5Impl* were classes that implemented the SHA1 and MD5 hash functions respectively. The user could not create instances of *SHA1Impl* and *MD5Impl* directly but would have used *HashFactory* instead. *HashFactory* had one static method, *selectMechanism*. This method returned the instance of a *HashMechanismProxy* class. This class implemented the same interface as all the hash functions. It was possible to extend the API with new hash functions similarly to TrueSign 1.1 API.

In the first version of this API, *HashFactory* returned pointers to *HashMechanisms*. During the design the author wanted to get rid of as many pointers as possible as they are inconvenient for the user. On the other hand, the author wanted to use polymorphism, otherwise the factory would have lost its meaning. It is not possible to return instances of abstract interfaces by value in C++ – that is why the pointers were used. A proxy class named *HashMechanismProxy* was implemented; it referred to the same *SingleHashFactory* interface as *HashFactory* and its instances created the necessary hash mechanism upon request. This was necessary to implement the copy constructor for *HashMechanismProxy* correctly (without a copy constructor the return-by-value mechanism cannot be used). The copy constructor was implemented so that only the pointer of *SingleHashFactory* was copied. In that case, instance A of *HashMechanismProxy* that was copied from instance B of the same class would not disturb the calculation process of B because both of them would be working in different contexts.

In addition to the usual hashing interface, *HashMechanismProxy* also had a method *hasImplementation*. It could have been that *HashFactory* did not have a specific *SingleHashFactory*. In that case, null pointer was given to *HashMechanismProxy* and no hashing could have been done.

HashFactory was an instance of Factory Method pattern, *HashMechanismProxy* used Proxy pattern. Both of those patterns are described in [GHJV95].

V. COMMENTS ON SOFTWARE DEVELOPMENT PROCESS IN TRUESIGN PROJECTS

Both projects – TrueSign 1.0 and TrueSign 1.1 had problems with quality and meeting deadlines. In this section the problems that occurred on the process level are examined. The author will not go very deep here since those problems have been investigated by Asko Seeba in his master's thesis [See01].

A group of people developing a software product is always applying some kind of software development process. The process unawareness of those people does not mean that they do not apply any process. In the worst case their process is not documented, steered and optimized which makes it impossible to make predictions about the process flow. This can be one of the causes for very long development cycles and unsatisfied customers. Those thoughts and previous experience with less clear process framework led the development team of Cybernetica to use the RUP framework to develop TrueSign projects.

There were two aspects of RUP forgotten in both TrueSign projects:

Firstly, RUP is architecture centric process. This means that before entering the construction phase there should be a clear idea about the structure of the system to be built. By structure it is meant that one must be able to identify the software architecture – the components of the system, their tasks, interfaces between and constraints on them. More detailed decisions should be pushed further in time. In TrueSign 1.0, the division into components was made but the tasks and interfaces of the components were left unspecified. Instead, a detailed class diagram of every component was given to developers. The same error was repeated in the TrueSign 1.1 project.

In TrueSign 1.1, the architects were excluded from the process in the beginning of the construction phase and were brought back a few weeks later. It is easy to understand that everybody needs a vacation from time to time, but it is not a good idea to leave the development team without architects in the phase where some concepts are still very novel and it is not clear what the outcome of the whole project will be.

Secondly, RUP is an iterative and incremental process. In iterative process, the software project is divided into iterations and at the end of every iteration one has executable programs that cover every aspect of the system to be built, although some aspects may be implemented with stub functions, without considering all the special cases, etc. Executable programs can be used to gather feedback from the client about whether the project is going to the right direction or not. The incremental part of the process means that at the end of every iteration the system has gained in functionality. Some functions that were merely stubs before are implemented now. At the end of the last iteration one has the whole system implemented.

TrueSign 1.0 failed to follow this guideline. There was only one iteration in the construction phase which is more characteristic to the waterfall process. This could also be named as one cause for late integration – in the beginning there was nothing to integrate because developers were busy implementing base libraries. The focus on the needed functionality was set too late in the process.

In TrueSign 1.1, three iterations were defined, which again seems to be somewhat too few for such a large project. There were also no validations of executable programs at the end of iterations.

A TrueSign 1.1 specific failure is that there was no inception phase before the actual development started. It was thought that the product would have all the

same requirements and risks as the technology demonstration, but this proved to be an error in the long run – the essence of the two projects was very different. One of the goals of TrueSign 1.1 was to produce a digital signature API. Little time was spent in analysis of existing APIs and no potential clients were queried about their wishes. From the developers' viewpoint it would have been helpful to have an active client by their side – someone who would actually try to use the API and make suggestions for improving it. Actually there was a client – a TrueSign client program was ported to the new API during the construction phase. Sadly, the potential of this collaboration was not used.

The experiments were not mentioned in this section. Both experiments were really small projects and both of them missed the most important stakeholder of the software development process – the client. There were no large teams and project management overhead in those experiments either.

RUP is described in [Kru98]. Besides RUP there are also other initiatives for development process - eXtreme Programming ([Beck99], [XP Web]) is an example of these. Some of the earlier documented development processes can be found in [Hos61], [Ben56] and [Roy70].

VI. DESIGN ACTIVITIES

1. Introduction

Software engineering professionals have invested plenty of resources into research issues related to the software development process. One part of the development process also includes actually designing the software. The author's opinion is that not enough effort is invested into researching the design workflow and design activities.

In this thesis the TrueSign projects have been examined from several viewpoints – how the process was guided, what means were used to communicate the design decisions during the development, which techniques were used to assure the quality of the design, etc. In this chapter several techniques that in some way were present in TrueSign projects are described and analyzed.

2. Communicating the design

The author's opinion is that software development is mostly about communication. The main problem with communication is restricted distribution of the knowledge, which in the worst case stays in just one place. If knowledge stays with the client, then the developing team has no way to learn what the client really wants; if the development team does not communicate with the client, the latter has no way of knowing whether the product that really was needed is received. If the project management is the knowledge holder then the development team and the client must work in darkness without knowing whether they are on time or whether their product is made at optimum cost. If one developer holds knowledge about one particular subsystem, then it will be hard for others to re-use and modify the subsystem; if architects and other

developers do not communicate then beautiful architecture can become spaghetti implementation or bad architecture can become a reality.

From the viewpoint of design communication, TrueSign 1.0 was an interesting project. Communication between project manager and developers was based on daily status reports; communication between architect and other developers was based on the poorly documented class diagrams.

In their essays, developers told that daily reports were useful to them as these made the developers to think about deadlines and organize their work. Poor documentation and unspecified requirements were considered to be a problem – most of the questions had to be discussed with an architect and it was hard to track down the reasons behind his design decisions.

Verbal communication between developers was encouraged also in TrueSign 1.1. The model had more useful documentation for developers and they were not required to waste their time updating it. Negative influence for the project was caused by the fact that the architects were not involved in the beginning of the implementation phase. Far too much hope was laid on the detailed documents where clarification of the concepts would have been more helpful.

3. Code reviews and pair programming

Code and design reviews are sometimes done by organizations to discover defects and possible shortcomings. In code review process, developers search the source code of one software unit for defects. The most usual approach, so called walkthrough, is that two developers read the same code together – one of them is the author of the code and another is the reviewer. During this session an additional pair of eyes can easily spot defects and questionable design decisions

that have found their place in the code. Code review can also be a good way to transfer knowledge and experience from developer to developer.

The technique where the code reviews are made a routine part of software design workflow as was done in the author's first experiment could be called conversational design. Two people work on the same part of the model or the source code while they are continuously reviewing each other's work.

Pair programming or pair design, as applied in TrueSign 1.1 elaboration phase and in the author's second experiment, brings conversational design technique to its extreme. Here, every line of the code or every method in the model is designed while two developers sit together behind one computer. One of those developers – the driver – has hands on the keyboard and does the typing job. The role of the second developer – the observer – is to help the first one think what to type and observe that no defects get into the code. The driver has more local insight of the designed method whereas the observer can think more globally about the design strategy. After some time (for example one hour), the developers switch roles.

Communication problems in TrueSign projects occurred often when developers started to modify each other's code. Here and there, some conceptual things were misunderstood or used incorrectly. Some subsystems were written by one developer in a way that was only understandable to the developer himself. Such problems could have been prevented with the help of code reviews or pair programming.

It is the author's experience that pair programming lays a great burden on the shoulders of developers. Both participants must be ready to communicate and well prepared before the session begins – the prerequisites of the task must be known to both developers. Otherwise at least one member of the party would feel quite useless in the beginning and the work would not be productive.

Pair programming implies concentrated and disciplined work. One cannot make 'e-mail pause' in every ten minutes, because one would be wasting the partner's time then. One cannot usually break the rules such as 'tests must be written first' because the other participant will deny the act. This phenomenon is called pair pressure in the literature ([WiK00]).

The positive sides to pair programming are:

- pair pressure – two developers working together are less likely to break project rules or to read e-mail instead of working;

- knowledge spreading – instead of one developer, both members of the pair know how the solution for the given problem works;

- error prevention and early detection – constant code review accomplished with pair programming helps to spot mistakes early in the process;

- team feeling – the problems with one developer (quality, health, attitude, etc.) will be noticed much earlier, the developers will feel more as a team.

The negative sides of pair programming are not so well known. The author assumes that the main problem with the technique is that it is very people-centric. The technique will be useful only if both partners are willing to apply it. Laurie Williams claims in her dissertation [Wil00] that certain personality types are less suitable for pair programming.

Pair programming is an interesting technique which, as a part of the development process, enforces communication between team members and helps to increase the quality of the software. Working in pairs does not result in development

cycles doubled in length. The studies ([WKCJ00], [CoW00], [WiK00]) have shown that the overall increase in project amount measured in man-months is 15%. Additionally, pair programming helps to ensure better quality of the product. Good introductions to pair programming are [WKCJ00] and [PP Web].

Pointers to a research on code review methods can be found in [SIRO Web].

4. Visual modeling

During the TrueSign 1.0 and TrueSign 1.1 projects, Cybernetica tried to implement RUP and use the development tools from Rational Software. One of the best practices suggested by RUP is to visually model software. "Modeling is important because it helps the development team visualize, specify, construct, and document the structure and behavior of a system's architecture. Using a standard modeling language such as UML, different members of the development team can unambiguously communicate their decisions to one another. Visual modeling tools facilitate the management of these models, letting you hide or expose details as necessary. Visual modeling also helps to maintain consistency between a system's artifacts: its requirements, designs, and implementations. In short, visual modeling helps improve a team's ability to manage software complexity." [Kru98]

From what the author has experienced, it is easy to describe the general structure and behavior of the software with the help of visual modeling language such as UML. Visual models are good from two aspects:

- representation – the design of a software can be represented in a way that is more understandable to human mind and the relations between software components become more visible than in the source code;

- abstraction – visual models let developers represent the design on a higher abstraction level than source code by leaving out unnecessary implementation details.

In the TrueSign 1.0 and 1.1 projects, visual modeling was used in two ways. Firstly the class-design was mainly made with the help of visual modeling tools, secondly those tools were used to regenerate the visual models from code. There were several problems with the tool used:

In TrueSign 1.0, several developers modified the model simultaneously. One of the project rules was that every change in class structure that occurs in source code must also be reflected in the model. As the modeling software used by TrueSign 1.0 team (Rational Rose [Rose]) did not allow simultaneous modification of separate model parts by different developers, every developer had their own copy of the model. Those copies were merged by one of the developers usually once a week. He spent several hours solving the conflicts and merging the models even though all developers worked on separate modules. As the model involves also the representation, a merging disaster will occur whenever two developers make slight changes to the layout of some model-based drawing. Those changes occur easily when two developers are using variable screen resolutions to view the model.

In TrueSign 1.1, most of the developers were freed from the burden of updating the model and one of the architects was given a task to reverse-engineer the model from the source code regularly. Still the tool support for visual modeling proved to be insufficient. Model updating took too much time and on the other hand the comments introduced to the source code by the tool made the code hard to read.

One of the dangers with visual models is that they can become out of date. The final product is not the model, but the source code. It can happen that in course of the project, the schedule becomes very tight and the developers will not update the model anymore, thus rendering the knowledge in a model obsolete and making the use of the model for making design decisions dangerous. The real problem is that visual model is a by-product of the development process – at one point one must start real coding. It is quite easy to forget real life constraints when designing software in UML. TrueSign 1.0 showed that the final design of the system was quite different in some parts from the design proposed by previously made models.

Also in TrueSign 1.1, the architects designed the whole API first in UML and then the actual coding was done together with other developers. Since the architects did not have all the necessary information at hand during their working sessions, several problems occurred during the implementation phase that also influenced the design.

In [Mey97], Bertrand Meyer introduces an approach where also the analysis of user requirements and modeling of the architecture is done with the help of abstraction mechanisms in programming language. Visual models are simply generated from the source code, thus enabling differently detailed views on the design. In this case there is no context switch from visual modeling to real programming and the model maintenance follows seamlessly. An alternative approach is taken in [ShM97]. Here the visual model of the system is specified in great detail and the source code is generated from it.

After TrueSign 1.0, some developers complained that the design models made prior to construction phase were insufficient and demanded for more detailed models to be made before any coding. Next time the architects gave their best to

produce more detailed models for programmers in TrueSign 1.1. The author thinks that this approach is not right. Software design is not only about abstractions, it is also about details and technicalities. Those details and technicalities compose a rather strong force that steers the design workflow. Most of the details usually get discovered during programming. If an idea is beautiful but cannot be implemented with current tools and mechanisms then this idea should be discarded.

Similarly to several software theoreticians, the author here draws a parallel between building a house and developing a software. When building a house, one can distinguish between at least two types of workers – engineers and construction workers. Engineers create plans that are used by construction workers to build the walls. Construction workers do not change those plans. In software projects, programmers are often seen as construction workers. They take the design models and build walls according to those models.

The author claims that developing a program according to design models is the task of an engineer and of not a construction worker. Design model for a house means plans that can be used by construction workers to lay bricks. Design model for software cannot be used to lay bricks – there are still many open questions that must be solved by engineering.

Those ideas and their conclusions are examined in [Ree92]. Same problems in a specific development process (eXtreme Programming [Beck99]) are discussed in [FowD00]. Some conclusions mentioned in [Ree92] are:

- Programming is a design activity – a good software design process recognizes this and does not hesitate to code when coding makes sense.

- Testing and debugging are design activities – they are the software equivalent of the design validation and refinement process of other engineering disciplines. A good software design process recognizes this and does not try to short change the steps.

- Many different software design notations are potentially useful – as auxiliary documentation and as tools to help facilitate the design process. However, they are not the software design.

5. Unit testing

In TrueSign 1.0, very few explicit steps were taken to assure the quality of the product. Every developer spent some time testing but one could not be sure whether all the functionality was tested or not. Usually, every software unit was tested with some of the most common input values and that was all.

The problems in TrueSign 1.1 started when some units' functionality, upon which the others would rely, was changed. The point was that there was no need to change the existing functionality but to add new functionality. To make it easier to add new methods, developers chose to alter the existing ones, which resulted in discovering defects in already tested modules.

To prevent such a situation from happening again, a unit testing framework was introduced in the middle of the project. By unit tests, those tests are meant that check the functionality of a software unit. These tests should get implemented with the unit and they are highly valuable as a documentation and quality assurance method. If a programmer must use a software unit, it is possible to read its unit tests which document the functionality and show how to use the unit. If the programmer must change the software unit to add functionality or refine the design for example, unit tests can be relied upon to discover defects that may

have been introduced to the code. Unit tests should be run regularly to inspect the software for recently introduced defects.

Unit tests should have the following properties:

- They must be automatic – the main idea is that the unit tests are executed as often during the project's lifetime as possible.

- They must be self-contained – if the developer checks a source tree out from the repository then everything needed for testing should be there. The tests must not require any additional files or directories, etc.

- They must be independent – if a unit test fails, it should be easy to track down the reason. That is why every unit test should have a clearly defined scope. If several variables commit to the success of a test then it can be hard to check which one of them really caused the test to fail.

- They must not require any user input – tests requiring user input make it hard to automate them, because the user must think what to enter and what consequences it may have on the testing session.

- The output of a test should only indicate whether the tests have passed or failed – in a bigger project there could be thousands of unit tests. It would be impossible to identify a failing test if every test would generate very much output. Please note that a test can generate output, but this output should be checked by the test itself. The user will not remember what the test was about even a few days after writing the test.

To make unit tests as beneficial as possible, a slight change to the micro-level design process was made in TrueSign 1.1 – unit testing was made mandatory and

unit tests were written by most of the developers most of the time. After creating a unit, a developer would also write tests for it and make sure that those tests would pass. Every time a developer would update local source tree from the source repository, all the unit tests should be run, thus making sure that changes from the source tree mixed with local changes did not break any existing functionality. Also, before committing any changes to repository, the developer had to check that the changes only added functionality, refined the design or fixed bugs instead of making existing things broken. Such an application of unit tests forms a safety net for developers where they can land into if they have unwillingly broken something.

The unit testing framework introduced in TrueSign 1.1 project turned out to be most useful during a follow-up project in August - September 2001 where TrueSign 1.1 API was ported from Linux to Solaris operating system. With the help of existing set of unit tests it was possible to measure the success of porting.

Still a unit testing framework should not be blindly trusted. It is possible that a poor set of unit tests leaves developers with the feeling that the software has no defects in it while at the same time the set of unit tests covers only 20% of the implementation.

Unit tests should not be confused with acceptance tests. Unit tests are written by developers for developers. Acceptance tests are written by developers and clients for developers and clients. Acceptance tests usually have their roots in use cases and their purpose is to verify that the system has all the functionality required by the client. Acceptance test can include user interaction, but they must be well documented. Good set of acceptance tests can be used as a tool to prove that the software does its task properly. The success of the TrueSign 1.1 API porting to Solaris was measured with the help of acceptance tests.

Besides unit testing and acceptance testing, developers must consider the need for integrating the software units. Large software systems consist of several units. Developers of those tend to make some implicit assumptions about the other units their own creation must interact with. To discover any false assumptions as fast as possible, the integration process for components should start early – in the end, the product is the system that consists of several units working together. This was forgotten in TrueSign 1.0 project. resulting in working overtime in the very end of the construction phase.

As a software architect, the author would like to suggest yet another group of tests – architectural tests that would provide the architects with the assurance that their ideas have been implemented properly. It is unlikely, though, that the correspondence between architecture and source code can be tested automatically. In reality, regular code reviews will have to be used in place of architectural tests.

The need for unit tests and acceptance tests is pointed out in [Hos61]. Most requirements for unit testing framework can be found in [Hos61] and [Ben56]. Both of these articles also mention the importance of planned unit integration process. There exist several tools ([XPSoft Web]) that make it easy to create unit tests and to run them automatically.

6. Refactoring

Several possible shortcomings can be seen in the design of the crypto subsystem in TrueSign 1.0 (parent class depending on the features of the child class, doubled RSA implementations, etc.). Those shortcomings make it hard to reuse this subsystem in a situation where there might be more crypto engines, signature schemes, etc.

In section II.6, it became clear that all those design problems could be solved by making simple structural changes in the source code without modifying functionality of the software. Such changes are called refactorings in the literature ([Fow00]).

- A refactoring is defined as a change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior [Fow00].

- To refactor means to restructure software by applying a series of refactorings without changing its observable behavior [Fow00].

By refactoring, the developer can clean up the design (before adding new functionality) in a way that makes it easy to add the new functionality. This ensures that the design will evolve with the changing requirements and will not become a bottleneck in the development process – the addition of the new functionality is not slowed down because of the inflexibility of the design. The new functionality is integrated with the existing structure and is not an artificial add-on to it.

Refactoring should only be done for a purpose – to ease the addition of new functionality or correction of the defects. If developers refactor without a need for it then they prepare the code for changes that may not occur – this is a waste of time. That is why the refactoring is done before coding and not after coding.

In TrueSign 1.1, one result of the project was an API that tried to solve much wider range of problems than the TrueSign 1.0 API. Most of those problems (such as possibility of unified interface for various keystores) are quite complex and require a deep understanding of different approaches. Such an understanding and experience is gathered over course of time. TrueSign 1.1 API is a result of a

revolution and not of an evolution. This also is a cause for plenty of its shortcomings. The author suggests to use the API in various products, to improve it and to keep it evolving. This way the necessary functionality and possible solutions can be factored out and a reusable API can be created.

Refactoring is explained in great detail in [Fow00]. Interesting information and tools for automating the refactoring process can be found in [Refactoring Web].

7. Test-first design method

In TrueSign 1.1, unit testing was made mandatory. It was the developer's responsibility and decision how to apply it in the design process. A design method that is rooted in unit testing and refactoring is called test-first design method.

The main idea behind this method is that one does not add a single piece of functionality without a unit test supporting it. The design strategy with test-first method can be described as follows:

1. Refactoring the code, if necessary.
2. Running all tests, the existing functionality must not be broken.
3. Writing a test to prove that the functionality to be added works as intended.
4. Compiling all tests, the test added in the previous step should not compile.
5. Creating necessary interfaces, adding empty method bodies.

6. Running all tests, the test added in Step 3 should fail, if it does not fail, then finding out why and then creating a test that would fail.

7. Writing the simplest possible code to satisfy the test and running all tests to make sure that no existing functionality was broken.

8. If it is possible to think of any more tests for this task then starting the whole process again from the beginning.

Test-first design consists of small iterations – test, code, refactor, test, code, refactor, test, code, etc. After every iteration, the functionality of the program is incremented. The programmer must think what to test, how to write the simplest code, what refactorings to do, etc., but the creative work of software design is under control with the help of this method.

The best side-effects of the test-first design are the tests. Although they do not prove that the code is correct and that new design decisions are not inconsistent with some previous design decisions, they still offer ground to believe that the program is functioning properly. It is also rather easy to code a method when there are some tests to execute it beforehand. Here the unit testing described in the section VI.5 really has its place in the design method.

When writing a function, the developer must think carefully about its inputs, outputs and side effects. If the developer wants to write tests for the function afterwards, the same process must be followed one more time. In test-first design method the process is made simpler. The developer of the functionality approaches the program from the user's angle. Developer as the user is aware of the inputs that may be given to the program and of the outputs that should be produced from these inputs. While writing a test, this knowledge gets documented and can be reused during the actual addition of the functionality.

This reduces the risk of forgetting to test the function's behavior on some occasions – the test exists before the functionality.

The functionality is what is really needed. The tests are just a by-product. If the functionality is implemented first then it is easy to forget the by-product – the client will probably accept the program if it seems to do what it has to do even though it has not been tested thoroughly. If one first implements the by-product, then after doing it one also must implement the functionality because this is what is really needed and just a by-product is not enough. Even if implementing the functionality is forgotten, the developer will be reminded about it next time the tests are run and as can be seen from the process description here, the tests get run quite often.

No design method known to the author denies the need for testing. Almost the same result can be achieved with iterative-incremental process where after adding some functionality the test is written. This approach requires more discipline, though. The developer must be dedicated enough to write the by-product after the real work has been done.

Test-first design tends to deal with technical problems earlier than other design methods. This makes sure that the design decisions can really be implemented but has the side effect that the design might become language specific. It is important to note that a good design on the lowest abstraction level takes the language into account as well. Although *HashMechanismProxy* from the fourth project is not necessary in Java, it is necessary in C++.

To write good software it is necessary to have a goal. In a typical software project this goal is architecture. Test-first design method as described here is not a way to construct the architecture, it is one of the possible tools to turn concepts in the architecture into an executable design.

It was easy for the author to think in the test-first way. The issues, such as the need for some abstract interfaces, came up naturally and if there was no urgent need for some class, then it was left out. There was also an ending condition for the task – coding is not done until all the tests pass. Also the feedback from the tests had a good influence on the author – it was good to know that some change in design did not break any existing code.

Test-first is not the only design method – some other methods are described briefly in the thesis as well. Applying of those methods will help to control the design process and to keep the creativity on the target. Working with no method or without understanding the method equals with not knowing how to solve a problem at hand.

Test-first design is a technique that is applied in eXtreme Programming (XP, [XP Web], [Beck99]) development process. A good example of test-first design method in practice is [Lip01].

8. Alternative design methods

There are other methods for design besides test-first method as well. This thesis deals only with those methods that the developer can directly apply to the personal coding process.

In [Wir71] Niklaus Wirth describes a design method called 'program development by stepwise refinement'. This method works by decomposing the original problem into smaller problems which are then solved by applying the method recursively. Firstly, the main program to solve the initial problem is written. All the subroutines are considered to be there and their implementations are not touched. On the next iteration the subroutines are written. Now their subroutines are considered to be there and their implementations are not

touched. So the program develops iteratively and on every iteration the data structures and subroutine hierarchy are refined using the following steps:

1. Writing the outermost level of the routine as if all the necessary subroutines were at hand.

2. Repeating the cycle for every unimplemented subroutine.

There is one problem with the approach – the code cannot be tested and even compiled before all the subroutines are implemented. This may result in the need to write all the code from scratch again.

Donald E. Knuth describes in [Knu97] the following mixture of composition and decomposition approaches for designing software units and/or larger programs:

1. Initial idea about how the code is going to be structured is generated.

2. A rough sketch of the program is constructed by decomposition approach as described above.

3. First working program. This time the cycle goes the bottom-up way. All the subroutines that do not call any unimplemented subroutines will be implemented.

4. Reexamination. The result of Step 3 should be an executable program. This program's structure is now looked through to spot any bad design decisions or possibilities for improvement. At this point the executable may be thrown away and the whole process will start again from Step 1.

5. Debugging. Here the program is executed with the debugger and its behavior is examined.

Efficiency of this method is not known to the author of this thesis and an analysis of this (or other design methods) could be a topic for further research.

All the design methods described in this paper operate on the lowest abstraction level of software design. With the help of those methods, the developer can produce the source code that can be compiled and executed. The notion used by those methods is the programming language and the heuristics or clichés of those methods are the heuristics or clichés of the programming language that base on the experience of past use of the language for solving a particular problem.

The definition of software design method is taken from [Bud93]. This book also gives a good overview of several software design methods that operate on higher abstraction levels than the source code.

VII. CONCLUSIONS AND FURTHER RESEARCH

In this part of the thesis the author shortly presents the main conclusions he has made from the described projects. Some proposals for future research are made also.

The main conclusions drawn from the work with TrueSign development team are following:

- **Design through coding** – the author believes that one should start the actual coding as early as possible in the development process. It should happen already during the phase of generating architecture – coding is the best way to try out the plausibility of architectural ideas. Anything that is better done while coding, must be done while coding. This includes

finalizing implementation specific details, designing the subsystems, specifying interfaces, etc. Coding is a design activity and should also be seen that way. Architecture may be looked at as a target for the design that is there to keep the developers on the right track. During the creation of the architecture, it is often hard to see the real problems and to define what really is the right track.

- **Communication** – Communication problems pointed at in section VI.2 should be paid more explicit attention to in software developing companies. When introducing a practice to software development method, one should spend some time thinking about the influence of the practice on the communication. Forced coding style rules may seem to be restrictive at first, but it really makes it easier to read the source code of the others.

- **Design method** – there has been a lot of research on the software development process; the amount of research related to design activities is inferior to it in the author's opinion. Design methods have been proposed e.g. in [Knu97] and [Wir71]. Currently, eXtreme Programming groups are elaborating on the definition of designing the software. The author thinks that design activities should not be addressed to as every developer's own business. There are several good practices and techniques that help the designer a lot. Those practices should be taught to developers and their use should be encouraged.

- **Measurement** – the problem with this thesis is that the author makes several claims but has not enough material to **prove** his words. This is also the problem of many development teams. TrueSign 1.1 team picked out RUP and started to implement it. They applied some methods that

seemed to work and discarded other methods that seemed not to work. Most of those decisions were not based on actual data gathered during the project but on the gut feeling of the developers. In order to say that some practice works, one should be able to measure the effects of it on the development process. The need for such measuring must be understood by the developers who actually gather the data and by the project management who should be responsible for gathering and analyzing the data.

- **Quality** – When talking about software quality, one can distinguish between inner and outer measures. The outer quality is what the client expects; it is the functionality and the look of the product. Inner quality is concerned with things visible to the developer – encapsulation, reusable code, easy to maintain implementations. The author's claim is that outer qualities do not come to life without inner qualities. A software with no structure or inflexible structure will not stand the test of time. To allow the software to change, one should continuously work on making it better – refining design when adding new functionality, re-thinking previous decisions, writing tests, etc.

The most important conclusions are as in [HuT99]: care about your craft and think about your work.

In the following some ideas for further research are proposed. The ideas are not novel but they represent the author's opinion on what is important in software engineering.

- **Experimental research on software design methods**. Software design methods in general have been analyzed in [Bud93]. Methods for

generating final design of the software product are described in [Knu97], [Wir71] and [XP Web]. One topic of interest would be to experiment with those methods and to analyze them. Experimenting in the author's opinion would include test groups who would apply various methods religiously and whose results would be evaluated and compared to results gained with help of other methods or with help of no method at all. The result of such an work would be better understanding of the methods which would allow the professionals to make more informed decisions while picking a method to apply in their development process.

- **Communication techniques**. The author of the thesis has started an experiment on pair programming with the intention to measure technical productivity of pair programming teams vs. conventional teams. It will also be investigated whether the results acquired with university students can be generalized to professional software developers. Similar experiments have already been done in North Carolina University ([CoW00], [WiK00]).

- **Communication problems in software engineering**. An analysis of several software projects from the communication viewpoint would be interesting. There are several types of participants in software project, all of them have hold on some kind of information. What happens if they do not get the information what they need, or if they do not communicate the information to each other? What symptoms would indicate what kind of disease? An interesting article about those problems is [Cop94].

VIII. RESÜMEE

TARKVARA KVALITEEDI PARENDAMISE MEETODID: JUHTUMIANALÜÜS

Sven Heiberg

Magistritöö

Tarkvaraarenduses põrkuvad sageli vastandlikud huvid. Ühelt poolt on tarkvara kerge muuta, tänu sellele omadusele on võimalik kiiresti reageerida klientide muutuvatele nõudmistele. Teiselt poolt vajab klient kvaliteetset ja töökindlat tarkvara, kuid pidevad muutused esialgsetes plaanides raskendavad kvaliteedi tagamist tunduvalt. Üheaegselt nii kvaliteedi kui ka paindlikkuse säilitamine on tarkvaraarendajate jaoks suur väljakutse.

Aastail 1999 – 2001 võttis autor Cybernetica arendusmeeskonna liikmena kahe suurema projekti raames osa pikaajalise tõestusväärtusega digitaalallkirja tehnoloogiat realiseeriva tarkvarasüsteemi loomisest. Autori vastutusel oli muuhulgas ka digitaalallkirja andmiseks vajalike funktsioonide teegi projekteerimine. Teegi projekteerimisele järgnenud realiseerimisfaas näitas mitmeid puudujääke teegi ehituses. Analüüsimaks erinevaid meetodeid kvaliteetse tarkvara loomiseks viis autor läbi kaks eksperimentaalset projekti. Esimese projekti käigus projekteeriti teek uuesti, teise projekti käigus rakendas autor testidest lähtuvat tarkvaradisaini meetodit (*test-first design method*).

Esimest ja teist projekti, vastavalt TrueSign 1.0 ja TrueSign 1.1, läbivaks veaks oli lootus, et tarkvara õnnestub arendada koskprotsessi põhiselt. Mõlemas projektis

läbiti kõigepealt projekteerimisfaas, mille käigus koostati detailsed spetsifikatsioonid kogu loodava tarkvara jaoks. Spetsifikatsioonide tegelik realiseeritavus jäi konstrueerimisfaasi tuvastada.

Võrreldes projektiga TrueSign 1.0 vähenes projektis TrueSign 1.1 arendajate ja projektijuhi vaheline suhtlus. Muuhulgas suurenes dokumentatsiooni osatähtus, samas kui otsene suhtlus tarkvaraarendajate ja arhitektide vahel jäi rohkem tahaplaanile.

Kahe eksperimentaalse projekti käigus proovis autor, inspireerituna tarkvaraarendusprotsessist eXtreme Programming, esialgsest mõnevõrra erinevat lähenemist arendusele, kasutades lähteülesandena väljavõtteid projekti TrueSign 1.1 ülesandepüstitusest. Kui kahes eelmises projektis osales suurem arendajate meeskond, siis eksperimentaalsetes projektides töötas 1-2 inimest. Sellest tulenevalt on ka magistritöö järeldused rohkem tarkvaraarendaja kui tarkvaraprotsessi kesksed.

Kõigis neljas projektis läheneti tarkvaraarendusele metoodiliselt. Rakendati tarkvaraarendusprotsessi *Rational Unified Process* (RUP) elemente ning erinevaid meetmeid loodava süsteemi õigeaegse valmimise ning kvaliteedi tagamiseks. Projektide kulg ning tulemused näitasid järgmiste aspektide olulisust tarkvaraprojektis:

- **Projekteerimine programmeerimise teel** – Autor leiab, et väga hea viis arhitektuursete ideede katsetamiseks ja kogumiseks on kodeerimine. Sagedaseks tarkvarasüsteemide mittetöötamise põhjuseks on liiga hiline programmeerimisega alustamine.

- **Suhtlus** – Paarisprogrammeerimine, koodi läbivaatused, regulaarsed raportid projektijuhile ja paljud teised suhtlustehnikad aitavad avastada

vigu arendusprotsessis ja loodavates programmides. Projektisisest suhtlust soodustavate meetodite rakendamine aitab kaasa ka projektiliikmete oskuste kasvule.

- **Projekteerimismeetodid** – Tarkvara kvaliteeti on võimalik parandada, kui arendajad teadvustavad vajaduse kasutada oma loovtöö protsessi struktureerimiseks läbiproovitud meetodeid. Kui projekti tasemel on vajadus metoodikate, nagu näiteks RUP, järele selge, siis ühe programmeerija tasemel kiputakse meetodi tähtsust alahindama. Magistritöös tutvustatakse kolme erinevat programmeerimismeetodit, põhjalikumalt peatutakse testidest lähtuval tarkvaradisaini meetodil.

- **Mõõtmine** – Tarkvaraprojektide analüüsimine ei ole võimalik ilma läbimõeldud ja juhitud mõõtmiseta. Uut arendusmeetodit rakendades või projektireeglit püstitades tuleb mõista, millist aspekti millises suunas antud meetod antud projektis mõjutama peaks. Sellisel juhul saab hilisemata tulemuste põhjal teadlikult otsustada, kas üldkokkuvõttes toodi projektile kasu või kahju.

- **Kvaliteet** – Tarkvara kvaliteet on mitmepalgeline mõiste ning omab erinevat tähendust tarkvara loojate ja tarkvara tarbijate jaoks. Samas ei saa tarbijale nähtav väline kvaliteet tekkida ilma, et arendajad kulutaksid aega neile nähtava sisemise ülesehituse kvaliteedi tõstmiseks. Magistritöö käsitlebki meetodeid, mis lihtsustavad tarkvara sisemise ülesehituse kvaliteedi tõstmist ning teevad seeläbi võimalikuks välise kvaliteedi loomise, kasutaja nõudmiste rahuldamise.

IX. REFERENCES

[ABRW01] A. Ansper, A. Buldas, M. Roos, J. Willemson, "Efficient long-term validation of digital signatures", *Proceedings of the Public Key Cryptography – PKC'2001 conference*, February 2001, pp. 402-415.

[ASN.1] *ITU-T Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1)*, 1993.

[Beck99] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[Ben56] H. D. Benington, "Production of Large Computer Programs", *Proceedings of the ONR Symposium on Advanced Program Methods for Digital Computers*, June 1956, pp. 15-27. Also available in *Annals of the History of Computing*, October 1983, pp. 350-361.

[Bud93] D. Budgen, *Software Design*, Addison-Wesley, 1993.

[CMS] R. Housley, "Cryptographic Message Syntax", RFC2630, 1999. Available online at www.ietf.org/rfc/rfc2630.txt.

[Cop94] J. O. Coplien, "A Development Process Generative Pattern Language", *Proceedings of the Pattern Languages of the Program Design conference*, 1994, pp. 183-237.

[CoW00] A. Cockburn, L. Williams, "The Costs and Benefits of Pair Programming", *Proceedings of the XP2000*, June 2000. Available online at www.pairprogramming.com

[cryptlib] P. Gutmann, www.cs.auckland.ac.nz/~pgut001/cryptlib/

[Crypto++] Crypto++, www.cryptopp.com

[Cyber Web] Cybernetica, www.cyber.ee

[Fow00] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

[FowD00] M. Fowler, "Is Design Dead?", *Proceedings of the XP2000*, June 2000. Available online at www.martinfowler.com

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gut99] P. Gutmann, "The Design of a Cryptographic Security Architecture", *Proceedings of the Usenix Security Symposium*, August 1999.

[Hos61] W. A. Hosier, "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming", *IRE Transactions on Engineering Management*, June 1961, pp 99-115.

[HuT99] A. Hunt, D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley, 1999.

[JBR98] I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1998.

[Knu97] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, 3rd ed., Addison-Wesley, 1997.

[Kru98] P. Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley, 1998.

[Law384] *Eesti Vabariigi Raamatupidamise Seadus (Riigikogu seadus nr. 384)*, 1994

[Lip01] R. Lipscombe, “Test First: Implementing an Arabic-Roman converter in C++, test first”, 2001. Available online at www.differentpla.net/~roger/devel/xp/test_first

[Lip99] H. Lipmaa, “Secure and Efficient Time-Stamping Systems”, PhD Thesis, University of Tartu, 1999, Available online at www.tcs.hut.fi/~helger/papers/thesis/

[Mey97] B. Meyer, *Object Oriented Software Construction*, 2 ed. Prentice Hall, 1997.

[MSCAPI] Microsoft, Microsoft CryptoAPI, msdn.microsoft.com

[OSSL] OpenSSL Project, www.openssl.org

[PKCS11] *RSA Laboratories: PKCS#11 v2.10: Cryptographic Token Interface Standard*, 1999. Available online at www.rsasecurity.com

[PKCS12] *RSA Laboratories, PKCS#12 v1.0: Personal Information Exchange Syntax*, 1999. Available online at www.rsasecurity.com

[PP Web] www.pairprogramming.com

[Ree92] J. W. Reeves, "What is Software Design?", *C++ Journal*, 1992. Available online at www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm

[Refactoring Web] www.refactoring.com

[Roo99] M. Roos, "Integrating Time-Stamping and Notarization", Master's Thesis, University of Tartu, 1999. Available online at www.cyber.ee/infoturve/research/publications/mroos-thesis.pdf

[Rose] www.rational.com/rose/

[Roy70] W.W. Royce, "Managing the Development of Large Software Systems", *Proceedings of the IEEE WESCON*, August 1970, pp. 1-9, Also available in *Proceedings of the 9th International Conference on Software Engineering* IEEE, pp. 328-338, (1987).

[RUP Web] www.rational.com/rup/

[See01] A. Seeba, "Unified Software Development Process and a Case Study of It's Application", (in Estonian) Master's Thesis, University of Tartu, 2001. Available online at www.ut.ee/~asko/tarkvaratehnika/magistritoo.pdf

[ShM97] S. Shlaer, J. S. Mellor, “Recursive Design of an Application-Independent Architecture”, *IEEE Software*, January/February 1997, pp. 61-72.

[SIRO Web] Software Inspections and Review Organization
www.ics.hawaii.edu/~siro/

[Sti95] D. R. Stinson, *Cryptography: Theory and Practice*, CRC Press, 1995.

[TrueSign Web] gns.privador.com
www.cyber.ee/research/eldorado.html

[UML Web] www.omg.org
www.rational.com/uml/
www.uml.org

[WiK00] L. Williams, R. R. Kessler, “Experimenting with Industry's "Pair-Programming" Model in the Computer Science Classroom”, *Journal on Software Engineering Education*, 2000. Available online at www.pairprogramming.com

[Wil00] L. Williams, “The Collaborative Software Process”, PhD Dissertation, University of Utah, 2000. Available online at www.cs.utah.edu/~lwilliam/Papers/dissertation.pdf

[Wir71] N. Wirth, “Program Development by Stepwise Refinement”, *Communications of the ACM*, April 1971, pp. 221-227.

[WKCJ00] L. Williams, R. R. Kessler, W. Cunningham, R. Jeffries, "Strengthening the Case for Pair Programming", *IEEE Software*, July/August 2000, pp. 19-25. Available online at www.pairprogramming.com

[X509] R. Housley, W. Ford, W. Polk, D. Solo, *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, RFC2459, 1999. Available online at www.ietf.org/rfc/rfc2630.txt

[XPSoft Web] www.xprogramming.com/software

[XP Web] www.extremeprogramming.org
www.xprogramming.com
c2.com/cgi/wiki?ExtremeProgrammingRoadmap