

Patrick Koch BSc

API Management and its Implementation in the Context of Agile Software Development

Master's Thesis

Graz University of Technology

Institute for Softwaretechnology

Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, January 2015

This document is set in Palatino, compiled with [pdfL^AT_EX2_ε](#) and [Biber](#).

The L^AT_EX template from Karl Voit is based on [KOMA script](#) and can be found online: <https://github.com/novoid/LaTeX-KOMA-template>

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____
Date

Signature

Eidesstattliche Erklärung¹

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____
Datum

Unterschrift

¹Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Foreword

In March 2014, I have started my work as student trainee at AVL and I was soon faced with issues about API management. This topic accompanied me right to the present. The first explanation for my responsibility in the department which I have experienced was to develop a so-called "interface checker". I have started programming the desired application and acquired a deeper knowledge as a result for this topic. After a few months, the idea arose to combine my work with a diploma thesis, to be able to gain more knowledge about API management and as a result from that, to improve my practical work at AVL. Moreover, I found that completing my masters program at AVL would be a pretty good idea and informed Dr. Peter Scheir who was my mentor at AVL about my wish. He liked the idea too and got down immediately to work for enabling my diploma thesis. I have to thank him for realizing my wish of a diploma thesis at AVL. Apart from that, he helped me a lot concerning this written paper, the corresponding practical work and in general, he gave me much helpful advice.

The next reference person, which I want to mention is Prof. Wolfgang Slany. He made this diploma thesis possible at the Technical University of Graz and for that he deserves many thanks too. It takes a lot of trust if a student wants to perform an external diploma thesis and I am glad to have reached this by working for him as study assistant for two years and by completing multiple of his courses.

Remains to mention, that I would not have achieved that level of quality of my work without my brilliant colleagues at the department: Stefan Preuer who always knew what should be done concerning to the programming. Dr. Harald Rosenberger, who made my tasks much much easier through the use of his pragmatic, goal-oriented thinking and last but not least Igor Roncevic who helped me too concerning the requirement and workflow capturing and effective testing for the application. Astrid Lock and Robert Korosec deserve my thanks too, for providing me useful input for this work.

Of course, a successful completion of a masters program is much easier with strong support of my mother and my father and of course my friends. Among my friends, I want to emphasize Sercan Akpolat who helped me a lot to steer in the right direction. This should not be underestimated and therefore I would like to thank them for their advices, patience and incentives.

Abstract

This diploma thesis is about theoretical aspects of API design, combined with agile development approaches, for improving the quality of APIs. Additionally, it consists of an expert interview, performed with experienced developers of AVL about API management and agile approaches and of a description of two C# applications, which I have programmed myself for detecting possible API breaks of an AVL product.

Designing high quality APIs represents often a challenge, because after a release further development must be taken with care. Triggering a break of an API is a horrible vision for any developer. There exists a series of best practices, for avoiding such a scenario but maybe greater support for that could be given through agile approaches. Therefore this work tries to find synergies between best practices of the development of APIs with agile approaches and to introduce C# applications which are detecting possible API breaks between different versions of a software product of AVL. The expert interview has the aim to identify those synergies and to get a confirmation of some best practices. So this work deals with detecting API breaks and determining how agile approaches could help to improve the quality of APIs.

Contents

Abstract	vii
Introduction	vii
1. API Management	3
1.1. Fundamentals and Motivation	3
1.1.1. Definition of an API	3
1.1.2. Contracts and Contractors	5
1.1.3. Benefits of Using APIs	5
1.1.4. When Should the Use of an API be Avoided?	6
1.1.5. Types of Compatibility	7
1.2. Attributes of a High Quality API	11
1.2.1. Abstraction and Key Object Modeling	11
1.2.2. Practical Definition of an API in C++ and C#	13
1.2.3. Information Hiding	14
1.2.4. As Small as Possible	18
1.2.5. Easy for Usage	20
1.2.6. Aspects of Coupling	23
1.2.7. Documentation and Testing	27
1.3. Practical Guidelines for Building High Quality APIs	28
1.3.1. Singleton Pattern	28
1.3.2. Factory Pattern	30
1.3.3. Proxy Pattern	31
1.3.4. Adapter Pattern	32
1.4. Maintenance of Backward Compatibility	33
1.4.1. Adding Functionality	34
1.4.2. Changing Functionality	35
1.4.3. Deprecating Functionality	36
1.4.4. Removing Functionality	37

Contents

1.4.5.	Versioning of APIs	37
1.4.6.	Branching	40
1.4.7.	API Reviews	43
1.5.	Testing	44
1.5.1.	Benefits of Implementing Tests	44
1.5.2.	Disadvantages by Writing a Huge Number of Tests	45
1.5.3.	Categorization of API testing	46
1.5.4.	How to Write Good Tests	48
1.6.	Conclusion	50
2.	Agile Software Development	53
2.1.	What is Agile Development?	53
2.1.1.	In Contrast to Agile Development: the Waterfall Model	54
2.1.2.	Key Aspects of Agile Development	55
2.2.	Agile Principles	57
2.3.	Comparison of Agile and Traditional Software Development	57
2.3.1.	TSDLC vs. ASDLC	58
2.3.2.	Advantages of Agile Practices	60
2.3.3.	Disadvantages of Agile Practices	62
2.3.4.	Advantages of Traditional Practices	62
2.3.5.	Disadvantages of Traditional Practices	63
2.4.	Conclusion	64
3.	Expert Interview	65
3.1.	Interview Structure	65
3.2.	Interviewees	66
3.3.	Interview	66
3.3.1.	First Part - Systems with APIs	66
3.3.2.	Second Part - Determining Best Practices	67
3.3.3.	Third Part - Review and Outlook	69
3.4.	Interview Answers	70
3.4.1.	First Part - Systems with APIs	70
3.4.2.	Second Part - Determining Best Practices	71
3.4.3.	Third Part - Review and Outlook	74
3.5.	Conclusion of the Statements of the Interviewees	76
3.5.1.	First Interviewee	76
3.5.2.	Second Interviewee	76

3.5.3.	Third Interviewee	77
3.5.4.	Fourth Interviewee	77
3.5.5.	Fifth Interviewee	78
3.6.	Conclusion	78
4.	API Development and Agile Software Development	79
4.1.	Positive Impact by Applying Agile Approaches	79
4.1.1.	Quick Feedback	79
4.1.2.	Cooperation of Customer and Developer	80
4.1.3.	Stable Contract	81
4.1.4.	Evolving API	81
4.1.5.	Refactoring	82
4.2.	Properties of a Well Designed API and their Relation to Agile Approaches	82
4.2.1.	Simple	82
4.2.2.	Documented	83
4.2.3.	Consistency	83
4.2.4.	Self Explained	84
4.2.5.	Reliability	84
4.2.6.	Integration of Adaptations only through Versioning	85
4.2.7.	Extendable	85
4.3.	Interactions of Agile Approaches and API management Based on Literature Research	86
4.3.1.	Collaboration of Developers with QA Engineers in a Scrum Sprint	86
4.3.2.	Determining Key Usage of an API	87
4.3.3.	Realizing Aspects of Coupling by Applying Design Improvement of XP	88
4.3.4.	Face to Face Conversation Applied in API Reviews	88
4.4.	Conclusion	89
5.	Tools for Determining API Breaks	91
5.1.	ApiDiff	91
5.1.1.	Term Definition	91
5.1.2.	Motivation	92
5.1.3.	ApiDiff in Collaboration with Artifactory	94
5.1.4.	Types of Reports	94

Contents

5.1.5.	Usage Possibilities	97
5.1.6.	Directory Hierarchy	101
5.1.7.	Testing	101
5.1.8.	Maintenance	103
5.1.9.	Automated TeamCity Testing	104
5.1.10.	Future Work	104
5.1.11.	Improvements according to Tests	105
5.2.	AssemblyParser	105
5.2.1.	Motivation	105
5.2.2.	Usage of AssemblyParser	106
5.2.3.	Integration of AssemblyParser in a Continuous Inte- gration System	107
5.2.4.	Future Work for AssemblyParser	108
5.3.	Conclusion	109
6.	Summary	111
A.	Interview Guide	113
B.	Interview Partners	115
C.	Examples	117
C.1.	Examples of Information Hiding	117
C.2.	Examples of Aspects of Coupling	121
C.3.	Examples of a Singleton Pattern	123
C.4.	Examples of a Factory Pattern	124
C.5.	Examples of a Proxy Pattern	126
C.6.	Examples of an Adapter Pattern	126
	References	129

List of Figures

1.1.	Structure of a modern application cf. (Reddy, 2011, p. 2)	4
1.2.	Warehouse of a car dealer cf. (Reddy, 2011, p. 22)	12
1.3.	Extended abstraction of a warehouse of a car dealer cf. (Reddy, 2011, p. 24)	13
1.4.	Implementation without a Manager class cf. (Reddy, 2011, p. 58)	26
1.5.	Implementation with a Manager class cf. (Reddy, 2011, p. 59)	26
1.6.	UML representation of a Singleton class cf. (Townsend, 2002)	29
1.7.	UML representation of a Factory pattern cf. (Purdy, 2002) . .	30
1.8.	UML representation of a Proxy Pattern cf. (Reddy, 2011, p. 91)	32
1.9.	UML representation of an Adapter Pattern cf. (Reddy, 2011, p. 94)	32
1.10.	Example of a "Major-Minor-Patch" numbering scheme cf. (Reddy, 2011, p. 242)	38
1.11.	Life cycle of an API cf. (Reddy, 2011, p. 249)	41
1.12.	Example of a branching model cf. (Reddy, 2011, p. 246)	41
2.1.	Waterfall model vs. Agile model cf. (Lotz, 2013)	55
2.2.	Agile Development cf. (Huston, n.d.)	56
2.3.	Traditional Software Development Life Cycle cf. (Shoukath, 2012)	58
2.4.	Agile Software Development Life Cycle cf. (Shoukath, 2012) .	59
5.1.	Current situation of the platform according to build break . .	93
5.2.	Process Sequence of ApiDiff in Artifactory Mode	94
5.3.	Artifactory Mode of ApiDiff	95
5.4.	Example of a Single Report	96
5.5.	Comparison of Interface Packages	97
5.6.	Example of an Overview Report	98

List of Figures

5.7. Directory hierarchy overview	102
5.8. Provided tests for ApiDiff, integrated on a TeamCity server	103
5.9. Batch file, responsible for building and running ApiDiff	103
5.10. Unit Test of ApiDiff	106
5.11. Integration of AssemblyParser in TeamCity	108
5.12. Batch script for starting AssemblyParser	108

Introduction

API management is a really interesting and also tough topic in software development, which could cause serious problems if anyone will not consider it with sufficient attention. Furthermore, the best knowledge about APIs may not lead to the success of a project, if no suitable development approaches are applied. This paper wants to determine how API development and agile approaches could be aligned together. This diploma thesis involves aspects of API management considering how agile development approaches can have a positive impact on API development, an expert interview and a description of two C# applications, which were developed for improving the quality of a product of AVL according to the APIs. The first part includes topics about design, maintenance and testing of APIs, therefore this chapter is essential for the understanding of APIs. The next chapter deals with agile software development, including a contrast to classical development. It is necessary to gain knowledge about agile approaches, for determining synergy effects to API development. In the third chapter, results of an expert interview according to API development and agile approaches with five experienced software developers of AVL will be published. This serves at the one hand to obtain a confirmation of some practices of the first chapter and on the other hand to serve as a reference for the next chapter. The fourth chapter explains how API development could benefit from agile approaches, by considering aspects of all previous chapters. In the last chapter, I will provide a detailed description of two C# applications, which were programmed for this thesis and which should serve to determine API changes between different versions of an AVL product. One of my main sources is "API Design for C++" by Martin Reddy ([Reddy, 2011](#)), nearly every practical code example is based on this book. I have converted some of the code snippets, which are all written in C++, to C# because this work should not focus on one specific programming language. Most of them were

List of Figures

kept in C++, because this programming language expresses more context related to API issues.

1. API Management

This chapter presents necessary basics based on literature research about how high quality APIs can be designed, tested and maintained, which are essential for the following chapters of the work. The main goal of this diploma thesis is to figure out how the approaches of this chapter can be optimized with agile ones.

1.1. Fundamentals and Motivation

1.1.1. Definition of an API

Martin Reddy describes an API thus: "An API is a logical interface to a software component that hides the internal details required to implement it" (Reddy, 2011, p. 4).

Jaroslav Tulach provides following description: "The API, in the sense useful for the clueless assembly of big system components, ranges from simple text messages to complex and hard-to-grasp behaviors of the components themselves" (Tulach, 2008, p. 36).

In general an Application Programming Interface (API) is based on the idea of providing an abstraction for a component, which solves a problem, by performing an interaction with it. In the best case an user of an API solves a task, for which the API was made for, as easy as possible and does not need to know the concrete process for solving that problem. It should just be clear how to use the interface, which hides the logic from the implementation. The software components are distributed as libraries, so they can be used in diverse applications.

1. API Management

According to this definition an API can be just a single function but also a bundle of multiple classes. The most important concept of an API is providing an intelligent interface which supplies functionality to other software. It is possible to build an application based on multiple underlying APIs, which are providing services for solving specific problems. It is well established, that modern applications are build in that way and also that an API may depend on another API.

For instance: An application for playing music depends on an API for loading music files and this API depends on an API for decompressing data.

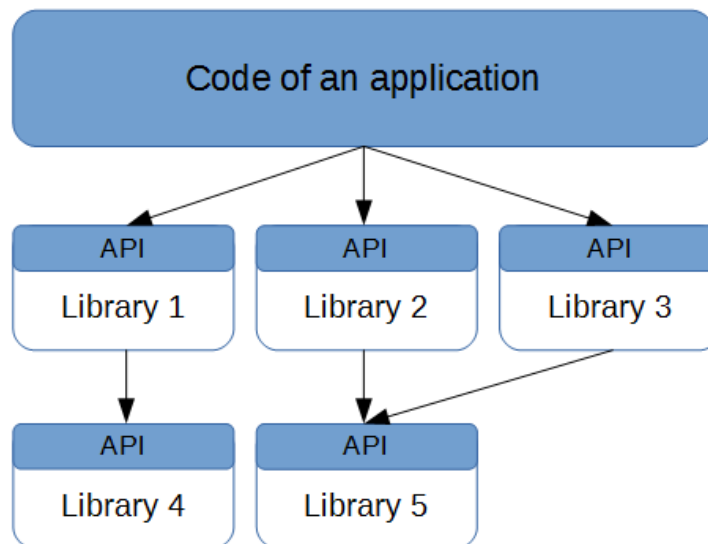


Figure 1.1.: Structure of a modern application cf. (Reddy, 2011, p. 2)

Figure 1.1 illustrates those dependencies. It shows an application, which has references to multiple libraries (represented by boxes in the picture), the interaction will be regulated through the APIs (represented by dark sections of the boxes). The brighter section of the boxes represent the concrete implementation, which should be burrowed (Reddy, 2011, p. 1).

1.1.2. Contracts and Contractors

The usage of APIs can be illustrated with a metaphor: considering the task of building a house, it is necessary to design it, to take care of laying bricks, electricity, water-supply, and so forth. This could be a huge effort, if a single person wants to do it on his own. A better way would be to hire an architect, and electrician, ect. who will take the specific tasks off and a probably much better way would be if some of those people are a friend or a colleague, so maybe they will do it for free. A contract with each contractor has to be prepared, according to which work has to be done and for which costs.

The process of building a house represents the compilation of a software application- there has to be take care of many different subtasks, in which a single person is not an expert. In this case a library with an API for the interaction has to be obtained. In the best case, this library is for free- which corresponds to a friend or colleague, who will help without payment (Reddy, 2011, p. 1-2).

1.1.3. Benefits of Using APIs

There are many reasons, which supports the usage of APIs:

- **Independent Implementation:** The implementation of modules, which are equipped with APIs improves the work a lot. A change of the implementation of a module does not imply that developers are either faced with some restrictions or that they will annoy their colleagues and clients.
- **Sustainable Development:** Without providing APIs, a software will become more and more fragile, because multiple components will have a coupling to each other. This leads to a so called 'spaghetti code'. Trying to refactor such a system, which grows during a long time, is a huge effort. A more sustainable way would be to invest in a good API design, for improving robust code. This will lead to a more transparent system with clearly defined responsibilities of parts of the system.
- **Encourage Modularization:** Each API should be related to a specific task. The introduction of APIs implies modularization, an application

1. API Management

which is build on the top of multiple modules. Those modules are independent of each other- this will also decrease the level of coupling.

- **Reduction of Duplicate Code:** If an API addresses a specific type of problem, then there will be no need for creating duplicate code. The functionality will be centralized and changing this functionality will affect all clients.
- **Code Recycling:** In the past, software development companies have written every functionality on their own. Nowadays, the development has become more modular: more external software is used in the implementation and many subtasks are outsourced. For example there exists a various amount of image reading APIs, which are used, tested, improved and maintained by thousands of people. For that, it is possible to focus on the main task.
- **Promoting Parallel Development:** In a team it is obvious, that a colleague may need some parts of code, which have been developed by another colleague and contrary. There is a need to have an intensive cooperation with each colleague, because some code may have a relatedness to other code. Then there will be an agreement on a contract- an API, so that nobody has to wait until the code, which is necessary from a colleague, is finished. This aspect is predestined for using a test driven development approach. Another advice is writing unit tests for proving the functionality, using an continuous integration software, which run all tests again and again. Is is mandatory to be sure that will no violation of the contract with a colleague.

As it can be seen, there are many benefits which will be received by using APIs (Reddy, 2011, p. 6-10).

1.1.4. When Should the Use of an API be Avoided?

Designing, testing, documenting and maintaining APIs are related to a bigger effort than handling the module without an interface. If a code will not be relevant for any client or colleague, than why implementing an interface? There exists the risk for writing sloppy code, in that case, applying the API design process will pay off. Indeed there are reasons for avoiding the usage of APIs:

1.1. Fundamentals and Motivation

- **Restrictions according to Licenses:** Using an open source library under the GNU General Public License (GPL) may lead to a problem, because in that case the whole code has to be provided for the public.
- **Implementation is not Available:** If a library is used, which is not open source, then the source will not be provided. This leads to troubles, which is straight forward when thinking of a bug in the library.
- **Lack of Documentation:** It may happen that the documentation of the API is not complete. This is a real problem if also no source of the API is provided.

In this cases represent possible restrictions according to the usage of APIs (Reddy, 2011, p. 10-11).

Other reasons would be the missing of a communication channel. Many companies provide such a channel to their customer, for notifying them about changes, improvements or problems of the API (Pedro, 2014).

1.1.5. Types of Compatibility

This chapter discusses the technical terms forward, backward, binary, source and functional compatibility related to versioning of software. Many aspects have to be considered for maintaining those compatibilities, which is especially very critical for APIs, because there is a direct impact to the clients of an API.

Backward Compatibility

Backward compatibility means, that a customer can switch to a newer version of the API without making any changes to his software. For that, the new API has to provide at least the same functionality as the old one. New functionality can be added, but no existing functionality can be changed without accepting incompatibility. The most important rule is: avoid withdrawals of the interface. Backward compatibility can be differed in:

1. API Management

1. **Functional Compatibility**
2. **Binary Compatibility**
3. **Source Compatibility**

Ensuring backward compatibility is a very important aspect to clients. By providing that, the trust of clients to the company will increase. This will of course improve the business value of the company (Tulach, 2008, p. 42).

To recap it in other words: an API supports backward compatibility if a client, who wants to use the next version of an API does not have to adapt any lines of his code (Reddy, 2011, p. 250-251).

Functional Compatibility

Functional compatibility is preserved if the behavior does not change by upgrading to a newer version. In this context, 100 percent functional compatibility can not be reached for all practical purposes. An example should confirm that: a simple (desired!) bug fix will (of course) change the behavior of the API, because a non desired behavior will be improved into a desired one.

This fact should be illustrated with the following C++ function of Figure 1.1: if a *NULL* is passed as argument and this character pointer will be accessed in the implementation of this function, then an error will emerge:

Listing 1.1: Simple C++ function, which needs a pointer as argument

```
int randomMethod(char* requiredParameter)
```

This bug will be fixed during the release of the next minor version. Generally this would improve the software, but functional compatibility would be rejected. If just the performance of the software is improved, then the functional compatibility is maintained (Reddy, 2011, p. 251-252).

Jaroslav Tulach defines functional compatibility in the same way: an API is functional compatible if the system, which uses the newest version of the API, delivers the exact same results as with the previous version of the API (Tulach, 2008, p. 49).

Source Compatibility

Source compatibility is reached if code can be recompiled against a newer version of the API without need for adapting the code. Neglecting the behavior of the software does not matter, it is just mandatory to compile and link it successfully.

By considering the first C# code snippet- see listing 1.2, it can be noticed that by changing the function, source compatibility will be retained because the second parameter is optional. In the example below, see listing 1.3, there is a need to find all incidents of the function if adaptations have to be made, because the second parameter has to be passed.

Listing 1.2: Code which maintains source compatibility

```
// version 1.0
public void RandomMethod(string requiredParameter)
// version 1.1
public void RandomMethod(string requiredParameter,
string optionalParameter = "default_value")
```

Listing 1.3: Code which does not maintain source compatibility

```
// version 1.0
public void RandomMethod(string requiredParameter)
// version 1.1
public void RandomMethod(string requiredParameter,
string nonOptionalParameter)
```

The main aspect of source compatibility is focused according the compilation of code (Reddy, 2011, p. 252-253).

In fact, maintaining this kind of compatibility is hard work, because adding classes, methods or removing them may lead to a breaking of the source compatibility. For that case, it should be considered to not put much effort to avoid source incompatibility for specific programming languages. At least this is a better solution, then never performing any necessary adaptation (Tulach, 2008, p. 43).

1. API Management

Binary

Binary compatibility, which is also called "Application Binary Interface Compatibility", is kept if it is possible to set an application, which is using version N of the API, to version $N+1$, by just relinking against the new API library. For that it is mandatory that the binary representation of all API elements (size, type, signatures of functions, alignment of structures) have to be persist. It is obvious that keeping binary compatibility is practically difficult, because changes according to the API will imply changes to the binary representation (Reddy, 2011, p. 253-255).

For a recap: Binary compatibility is therefore maintained if every application, which could be linked successfully in the past, could still be linked (Tulach, 2008, p. 49).

Forward

If client code, which compiles against a newer version of the API $N+1$, compiles against the API version N too, without the need any changes in the code, than forward compatibility is realized. A more convenient explanation reads as follows: Adding new functionality will cause a break according to forward compatibility (if the changed API functionality is used by client code), because it is impossible to downgrade to an older version.

In the first C# code example- see listing 1.4, forward compatibility is not broken: client code will compile against the older version when downgrading from 1.1 to 1.0, because the second parameter is optional.

Listing 1.4: Code which maintains forward compatibility

```
// version 1.0
public void RandomMethod(string requiredParamter,
string optionalParamter = "default_value")
// version 1.1
public void RandomMethod(string firstParamter,
string addedParameter)
```

A counterexample is shown in the next C# code snippet, see listing 1.5: the second parameter of version 1.1 is not present at version 1.0. A client

1.2. Attributes of a High Quality API

specified code, which compiles against version 1.1 will not be compatible with version 1.0.

Listing 1.5: Code which does not maintain forward compatibility

```
// version 1.0
public void RandomMethod(string requiredParameter)
// version 1.1
public void RandomMethod(string firstParameter,
string optionalParameter = "default_value")
```

By considering those examples, a common problem is faced: a method needs to be changed and therefore e.g.: a new parameter has to be implemented. If this is realized, then the forward compatibility is broken. To reach this type of compatibility, a developer needs more forethought about the functionality. For that, a new potential parameter of a method could be planned with this technique as shown in listing 1.4. The parameter will not be necessary for the current version, but for future versions. Just highlight the variable as unused respectively optional, for instance with naming it "unused" or "optional" (Reddy, 2011, p. 255-256).

1.2. Attributes of a High Quality API

This chapter is about defining quality standards for an API, e.g.: hiding information, consistency and loose coupling.

1.2.1. Abstraction and Key Object Modeling

An API is defined and developed to solve a specific problem. For that purpose, an abstraction of the problem has to be derived, which the API provides. It is very difficult to create a good abstraction. It should be reached that a person, who does not possess a technical background, is able to understand the abstraction and the concepts of the API according to the problem, just by providing the documentation. To be more precise, well

1. API Management

defined abstractions and concepts have a positive impact to the consistency. For that, every class with its methods should correspond to one specific purpose- this concerns to the name of the class, methods, members, ect. too.

The practical example of an auto dealer, as seen in Figure 1.2, should clarify the concept of an abstraction: A list of all cars is administrated, which contains all details of each car. So, there is a need for a *CarList* object. A *Car* instance, illustrates the name, engine power, type of drive, version, ect. of a car. If a car delivery is made, the car has the be added to the list. In case of a sale, the specific car has the be removed from the list. Those two methods belong to the object *CarList*.

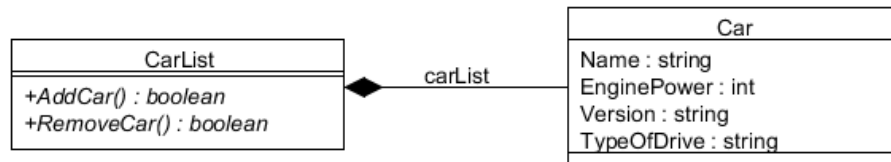


Figure 1.2.: Warehouse of a car dealer cf. (Reddy, 2011, p. 22)

This simple abstraction fits at the moment, but for the purposes of extensibility, it should be possible to offer a car with diverse version variants, e.g.: an estate wagon or an compact version. One possibility would be to add a second version member to the *Car* class, e.g.: `Version2`. This is of course not a good solution, because there exist maybe no second version for a car. It is now necessary for modeling the key objects of the specific problem. This process is called "object-oriented design" or "object modeling". The aim is, to identify all important objects and their corresponding methods of the problem. This process should be driven according to the specific properties of an API. Those requirements would be:

- Each car could have multiple versions
- A car list could have multiple cars with the same name
- A data set of the car list could be edited

1.2. Attributes of a High Quality API

- Each car could have multiple types of drive (diesel, petrol)

To fulfill those properties, the abstraction has to be improved:

- Every car holds a list of *Version* objects (estate wagon, compact)
- Every car holds a list of *TypeOfDrive* objects (diesel, petrol)
- Every could be identified by an unique number

By implementing those requirements, the new UML diagram looks like as in Figure 1.3. Obviously, the API design has to be adapted by every new functionality. It would be a good piece of advice to review every design and planning ahead to possible new functionalities. There exists no "best" modeling, there are always multiple possible ways for solving a problem (Reddy, 2011, p. 21-25).

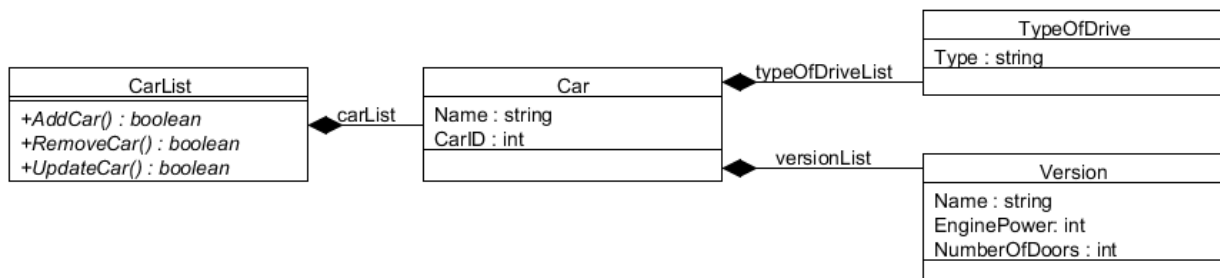


Figure 1.3.: Extended abstraction of a warehouse of a car dealer cf. (Reddy, 2011, p. 24)

1.2.2. Practical Definition of an API in C++ and C#

As previously mentioned, an API embodies an abstraction to a software component and specifies how to interact with it. In C++, it can be described as one or multiple header files (.h). Additionally there exists documentation files. The corresponding implementation is in many cases available as library file, which can be linked to the applications of the user. Examples would be static (.lib) or dynamic (.dll) libraries on Windows, ".dylib" files on Mac or .so files on Linux.

In C++ an API consists of:

1. API Management

- **Header Files:** Multiple ".h" files, which are responsible for the definition of the interface. Without including those headers, it is not possible to compile user generated code against the interface. In case of open source APIs, the source code is provided too (".cpp" files).
- **Libraries:** Represent the implementation of the API in the form of dynamic or static library files (e.g.: ".dll" files).
- **Documentation Files:** A documentation, which describes the interaction of the API

This defines the practical definition of an API in C++, this suits of course not to every programming language (Reddy, 2011, p. 3-4).

In general, the same applies to C#, but there are no header files necessary: the definition and implementation of interfaces can be found in the corresponding .cs- files.

1.2.3. Information Hiding

This is probably the most important property of an API: the user of the API does not gain an insight to the implementation. This implies, that the programming logic can be changed, without disturbing any user. Information hiding can be differentiated between physical and logical hiding. The physical way is applied by simply not providing the source code to the user. The logical way on the other hand limits the access by special features of the programming languages (Reddy, 2011, p. 25).

Physical Hiding

As already mentioned, physical hiding is executed by avoiding to deliver the source to the client. At first, the difference between a declaration and a definition has to be cleared:

A declaration determines just name and type, no memory will be allocated, as seen in Figure C.1. Contrary, a definition requires an allocation of memory, according to C.1, an example of a definition can be seen in Figure C.2.

1.2. Attributes of a High Quality API

In the C++ development, declarations are provided in the header files (.h), while their definitions can be found in the source files (.cpp). According to the C++ conventions, it is also possible to perform a definition in a header file, as seen in Figure C.3: the compiler is forced to inline the method *printName()* at every line of code, where it is called. This is a bad practice of API design, because the code will be unmasked. In general, header files should only be used for providing the declarations (Reddy, 2011, p. 25-26).

In C#, declarations and definitions can be found in the appropriate .cs files.

Logical Hiding

In object-oriented programming, the concept of encapsulation is used for limiting the access to members of classes or structs.

- **Public:** unrestricted access
- **Protected:** access is possible to members of derived classes
- **Private:** access is only possible for members of this class

The concept of encapsulation can be used for logical hiding. If there are no API boundaries, an user will just try to get what he needs, for finishing his job. Sounds not for a problem, it might be one if the software has to be maintained or adapted. It will get more complicated, because external code has to be considered too (Reddy, 2011, p. 26-28).

It is recommended for keeping the level of access to classes, methods and members as small as possible. This can be done by setting as much as possible to private, for maximizing information hiding. Not only information hiding will be improved, it will have an impact to the coupling of an API too, because as more classes, methods and members are not public, as more can be easily changed without affecting the API (Bloch, 2007).

Providing Getter and Setter

Encapsulation refers also to a concept, in which members and their corresponding methods are in a bundle. According to a good API design,

1. API Management

members of a class should never be declared as public. The advice would be to implement getter and setter methods.

A bad C++ example can be seen at Figure C.4, in which all member variables are declared as public. For improving this example, getter and setter methods have to be implemented, the result can be seen in Figure C.5, in which all member variables are set to private and corresponding getter and setter methods are implemented. This implies more effort, but it is in the long term a better solution, if changes have to be done. There exists a lot of benefits according to this usage:

- **Validation of Internal State:** Verification of the arguments, before assigning them to members.
- **Caching of Frequent Values:** Values, which are frequently requested, can be stored and returned.
- **Debugging:** Insertion of logging statements, so it can be determined which members are accessed.

Getter and setter methods imply more programming work, but they improve the quality of an API (Reddy, 2011, p. 28-30).

The disadvantage of just using a member without the corresponding getter and setter is obvious: there is only a read or a write access possible to the member. By using a getter, additional actions like a lazy initialization or an access synchronization can be done. Providing a setter enables for instance the possibility for implementing a notifier, e.g.: if the value changes (Tulach, 2008, p. 70).

Hiding Member Variables

If member variables are not part of an interface, then they should not be visible to the client. A bad example for that can be seen in Figure C.6, in which the public member *theStack* could lead to a downfall. The member variable *theStack* is realized through a simple integer array. There are corresponding functions like *topOfTheStack()*, which returns the current item on the top of the stack and the member *currentSize*, which determines the size. A client may make use of *theStack*, for instance by a direct access. If

1.2. Attributes of a High Quality API

this variable will be substituted by a list or a vector because of a changing implementation, then all users who are using *theStack* get a problem. This will break a user specific implementation. For this reason, hiding the member variables will be a sustainable solution, as shown in the improved version of the stack example in Figure C.7 (Reddy, 2011, p. 30-31).

In general it is a good advice that no public class contains any public member. The exception for that rule are constants (Bloch, 2007).

Hiding Methods for Implementation

Hiding just the members of a class is maybe not enough. It is a good practice to do not exposing the methods, which are not part of the interface too. An example class *RetrieveCertificatesOfX509Store* as seen in Figure C.8 should allow an user to retrieve a certificate of a X509 store. At the first view, everything seems to be alright, the member variables are private but several methods are made public, for instance a method which will open a X509 store. A client should not be able to use this method, the method for retrieving certificates by passing a X509 store is sufficient and it is the aim of this class. The method *getStoreName()* involves the risk that an user will get access to a private member of the class and tries to manipulate the current state. Another bad example would be if access to a non constant pointer or a reference of a object of a private member could be achieved. In this case the current state of the class can be modified too, without using the API. The obvious solution in this example would be to set all methods (except the constructor and the method *RetrieveCertificates*) to private. It is possible to hide the methods from the compilers point of view, but a human could inspect the methods too, because the header files have to be provided by an *#include*. That is a constraint of C++: all members (private, protected, public) in the declaration of the class have to be provided. There exists one common practice, called "Pimpl idiom" for separating the private members from the public header files. This is solved by a pointer, which references from the header file to the implementation class. Another solution would be to move the private methods from the header to the source files (.cpp). Then they have to be converted to static methods- this is possible if the

1. API Management

private methods only accesses public members of the class (Reddy, 2011, p. 31-33).

Hiding Classes

Hiding a whole class would be another possible approach, for instance if the class contains only implementation specific code. Not every class has to be public, as seen in the C++ example of Figure C.9: their purpose is to provide a small picture show, with multiple pictures, which appear by an glide through the desktop of an user. It is possible for determining how many pictures will appear, the speed of them, ect. For that, a knowledge about every movement of each picture and for updating their state is required. This is solved by a separate class *SinglePicture*. A client does not need to know anything about this class, because the usage of the API of the class *PictureShow* is sufficient. For this reason, the class *SinglePicture* is set to private (Reddy, 2011, p. 33-34).

1.2.4. As Small as Possible

On one side an API should be minimized whilst on the other side it has to be designed to serve its purpose. Determining the happy medium could be very tricky, because the boundaries of the API may become indistinct. If there exists a doubt about e.g.: a class or a method in that context, then it should not be implemented. This rule serves as a rough guide. It is a better way to implement less in the case of an unsureness, because if it is implemented, clients or colleagues can use it. From that moment, it has to be maintained (Bloch, 2007).

Danger of Providing Virtual Functions

Inheritance of an API by providing virtual functions, should be used with care. If this is performed, clients can use their own implementation of the methods in the subclasses. This may lead to a bundle of problems:

1.2. Attributes of a High Quality API

- **The Fragile Base Class Problem:** Small changes in the base class may imply unimagined influence to the client. This can happen because the base class is developed in isolation. It is unknown how the client uses the API. This phenomenon is also called "fragile base class problem".
- **Breaking Internal Integrity:** Overridden methods may cause a break of the internal integrity. Virtual methods of a base class may call other methods of the class by default and this should not be forgotten in the overridden ones.
- **Error Prone Extension:** Clients can extend the API in an error prone way: An API with multi-threading support in combination to a client's implementation of overridden methods with no locking discipline will lead to a problem.

To circumvent those problems: enabling the possibility to override should be used carefully. Virtual functions should only be implemented if it is sure that it will not lead to any of these kind of problems. In general a class without virtual functions is more robust and it is easier to maintain. Considering following rules, the creation of subclasses should be allowed:

- Providing a virtual destructor, so that clients can clean up with allocated items.
- A documentation is mandatory, which explains the calling hierarchy of methods. If a client wants his own implementation of a method, he has to know which methods have to be called.
- Virtual functions should never be called in a constructor or a destructor because those calls will get lost according to a subclass.

In general, the usage of virtual functions must be treated with caution ([Reddy, 2011](#), p. 36-37).

It is recommended, to forbid the creation of derived classes by clients. This works of course just for classes: it is desired that clients are able to create their own implementations by providing an interface. If a class and not an interface is involved, it is quite likely, that clients will derive subclasses, if it is possible. There exists a risk for non virtual functions too, because they could be overridden in a derived subclass, which enables a different behavior of the function. This leads to the problem, that the API has to support those varied interpretations. It can be solved by not providing

1. API Management

a public constructor (in the next chapters, a Factory method should be preferred against a constructor), but the most effective way would be to restrict subclasses of non interfaces (Tulach, 2008, p. 73-74).

Convenience APIs

As already mentioned, an API should be kept as small as possible, but there occur strains when reducing the number of functions and simultaneous improving the usage of it. Developers are facing a dilemma: should they provide convenience wrappers (summarizing multiple routines for performing higher level operations) or not? This would provide multiple ways for solving a problem. On the one hand, there should exist just one way in an API to perform an operation, this keeps the API stable and supports the usage. On the other hand, customer should not need to write much code on their own for executing some tasks- which would lead to writing boilerplate code.

For circumventing this dilemma, it is mandatory to not mix the convenience APIs with the core API. Establishing supplementary classes, which wrap specific functions of the core API and separate it from the core API (different files) would be a solution. For that, the convenience API has a dependence to the core API (Reddy, 2011, p. 37-39).

1.2.5. Easy for Usage

An API has reached an adequate level of simplicity if there is no documentation necessary for using it. A client should be able to get the purpose of e.g.: a method, by just inspecting the signature. This should not be an excuse for not providing a well structured documentation. The next topics should help in making the API more easier to use (Reddy, 2011, p. 39-40).

- **Self-explanatory:** An API has a high degree of discoverability, if it is possible to use it without any documentation. There are several ways for performing that: Realizing a logical object model, providing meaningful names for classes and methods and avoiding abbreviations like

1.2. Attributes of a High Quality API

getNumOfDifMet() instead of *getNumberOfDifferentMethods()* (Reddy, 2011, p. 40).

- **Avoiding a misuse:** No multiple interpretations of the usage of a method or a potential misuse of a method should exist. Following simple function, as seen in Figure 1.6, should give more context for that advice:

Listing 1.6: C++ Function for a potential misuse

```
bool existsFileInDirectory(char* filename, char* directory,
    bool inRootOnly, bool caseSensitive);
```

The first and second parameter are providing the name of a file to search for and a directory name. After them, a boolean value is passed, which indicates whether there should be searched in the root of the directory only. The last one determines, whether the search should be applied case sensitive or not. It is obvious, that the last two arguments may be mixed up. This would lead to different results when using this method. Two simple enums, as seen in Figure 1.7 solve that problem. By implementing this, a misuse of this method becomes more difficult.

Listing 1.7: The usage of enums may improve the usability of APIs

```
enum Locations {
    ROOT,
    ALLSUBDIRECTORIES
};
enum Sensitivity {
    CASE_SENSITIVE
    CASE_INSENSITIVE
};
bool existsFileInDirectory(char* filename, char* directory,
    Locations location, Sensitivity caseSensitive);
```

By using enums instead of boolean primitives, the code becomes more readable too. In some cases another type than a enum would be needed- the implementation of classes should be to reconsidered (Reddy, 2011, p. 40-42).

1. API Management

- **Consistency:** The whole API should be kept consistent according to naming conventions, error handling, throw of exceptions, argument order, ect. E.g.: if a decision was made to use the word pair "new" and "old" then there is no switch possible to "current" and "old". This might lead to confusion. Two copy functions of the Linux man page serve as a bad example: *cp*, which copies files and directories and *strcpy*, which copies a set of character. Both use different abbreviations for "copy". According to the consistency of parameters of functions, an observation of *malloc* and *calloc* would be interesting. In general, those two functions have the same purpose, as seen in Figure 1.8: they allocate memory (*calloc* includes an initialization with zero bytes), but both methods have a different calling convention- this is of course a bad example according to consistency.

Listing 1.8: Calling convention of the function calls 'malloc' and 'calloc'

```
void *calloc(size_t count, size_t size);  
void *malloc(size_t size);
```

Consistency should be applied on class level too.(Reddy, 2011, p. 43-45) It is very important, that this specific concept, which has to be acquired by the user of the API will be the same through the whole API. Another example would be a registration of factories of objects. If this was introduced, then every type of factories has to use that specific registration process (Tulach, 2008, p. 38).

- **Avoiding a Long Parameter List:** In the best case, an user just has to pass fewer or equal than three arguments. In the other case, they have to look into the documentation, but an API should be used without a need to refer to a documentation. If more than three parameters are necessary, the method can be broken up, or a helper class can be implemented for storing the parameters (Bloch, 2007).

In addition to all that, a metaphor should demonstrate the relationship between the creator of an API and its clients. It should be pointed out again, that users of an API have to understand it. That is a very important aspect. Jaroslav Tulach compares the development of an API with writing a book: There is just one author but multiple clients of the book, who want to read it. The author of the book has to estimate the skills of his readers, because in contrast to his readers, who may know much about him- the author does

not really know much about them. For that purpose, writing a book is an art- like developing an API (Tulach, 2008, p. 37).

1.2.6. Aspects of Coupling

Design guidelines of software systems suggest to provide a low coupling with a high cohesion. Coupling means to decrease the dependence between different software components, cohesion is a measure of connectedness between functions of a single component. In summary, that is an approach to improve the encapsulation and the independence of software components. An alternative explanation would be to imagine two components: *componentOne* and *componentTwo*. How does *componentOne* has to change, if *componentTwo* changes? There exists a set of indicators, which might help to determine the degree of coupling:

1. **Size of Coupling between Components:** How many classes, methods, parameters of a methods exists according to the connectedness between components? E.g.: a method with three parameters, which is called by another component, is stronger coupled than a method with two parameters.
2. **Visibility:** Changing a global variable for modifying the current state of another components has a low degree of visibility.
3. **Intimacy:** Determines the degree of directness of the connectedness according to the components: components can be directly coupled and indirectly, e.g.: *componentOne* has a couple to *componentTwo*, which has a couple to *componentThree*- in this case *componentOne* and *componentThree* are indirectly related to each other. Another example would be inheritance in comparison with a composition ("has a" relation): the inheritance is stronger than the composition, because the inherited class has access to all members of the base class. Contrary there is only access to public members possible.
4. **Flexibility:** What is the level of flexibility of a code? A method of *ObjectTwo* has to be called from *ObjectOne*, but the passed parameters have to be adapted: is it necessary to make many changes in the code?

1. API Management

Bad Practice:

It should be avoided to design components, which are directly or indirectly dependent from each other: if this would not be beware, a component could not be used independent from other components. A reuse is not possible too (Reddy, 2011, p. 52-53).

Following practices may help to decrease the degree of coupling:

Using Forward Declarations

In some circumstances, two classes can be decoupled by using a forward declaration instead of implementing the interface. This works e.g.: if one class does not need to call methods of the second class, as seen in Figure C.10. *ClassTwo* just requires a pointer of *ClassOne*. There is also no need to use the *#include* statement. So the advice is to use a forward declaration, except it is necessary to use the whole interface (Reddy, 2011, p. 53-54). Forward declarations can not be applied in C#, because they are not supported in this language. The order of declarations in C# does not matter in contrast to C++ (Robinson et al., 2004).

Preferring Non-Member Functions

If it is possible, then an implementation of member or friend functions should be avoided. This has a positive impact to encapsulation and functions will be more loosely coupled to the class. Figure C.11 shows a C++ class, which consists of member functions only. The method *printAdress()* has access to a private member *address*. Under this circumstances, a coupling exists between this function and the private member variable. This coupling can be solved by the implementation seen in Figure C.12: Now the *PrintAddress* function requires an instance of the class *MyClass* as argument. This implies that this method can only get access to public members any more (Reddy, 2011, p. 54-55).

Redundant Code is Better than Coupling

In some cases, redundant code can solve tight coupling problems. Preventing redundancy implies that code will be used by several components, but this increases the degree of coupling. Two components might have a coupling, because one component needs a functionality of the other one. If this functionality can not be separated that both can access it, then providing some redundant code might help. . In Figure C.13, a simple shopping basket is implemented. The method *AddProductToBasket* expects a reference of an object of the type *Product* and an integer as arguments. As it can be seen, there is a coupling between the class *ShoppingBasket* and *Product*. The implementation of the method *AddProductToBasket* can be adapted in that way that the name of the product has to be passed instead of a reference of an object. This name can be accessed by *Product.GetName()*. This will decouple the two classes, as seen in Figure C.14. Instead of referencing the class *Product*, just a string has to be passed. As a side effect, the *nameOfProduct* member is now implemented in the class *Product* and in *ShoppingBasket* (Reddy, 2011, p. 56-57).

Using a Manager Class

Manager classes are used for coordinating multiple lower-level classes. This has the advantage that dependencies between several lower-level classes will be broken, as seen in 1.4. This simple UML diagram shows a device manager, which is responsible for managing different devices. If there is no manager class provided, there will be an interlacing according to the dependencies. It is possible to bring in more structure with introducing a manager class, as seen in 1.5. The class *DeviceManager* takes over the administration between *DiverHandling* and *Deinstallation* (so they just have to depend on *DeviceManager*) classes and the devices (Reddy, 2011, p. 58-59).

1. API Management

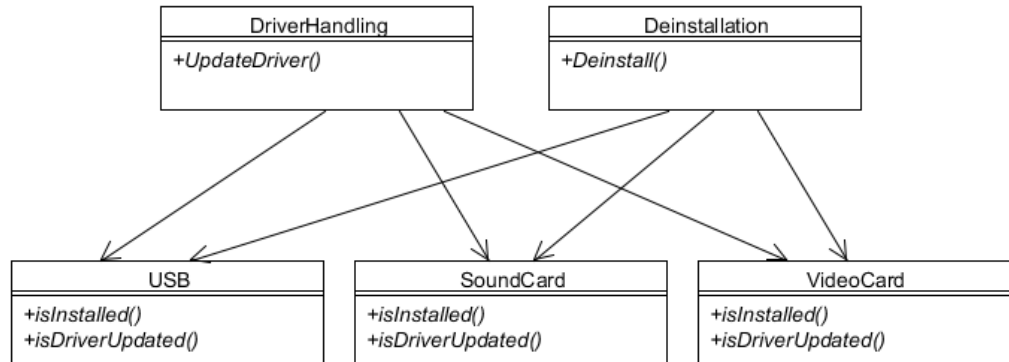


Figure 1.4.: Implementation without a Manager class cf. (Reddy, 2011, p. 58)

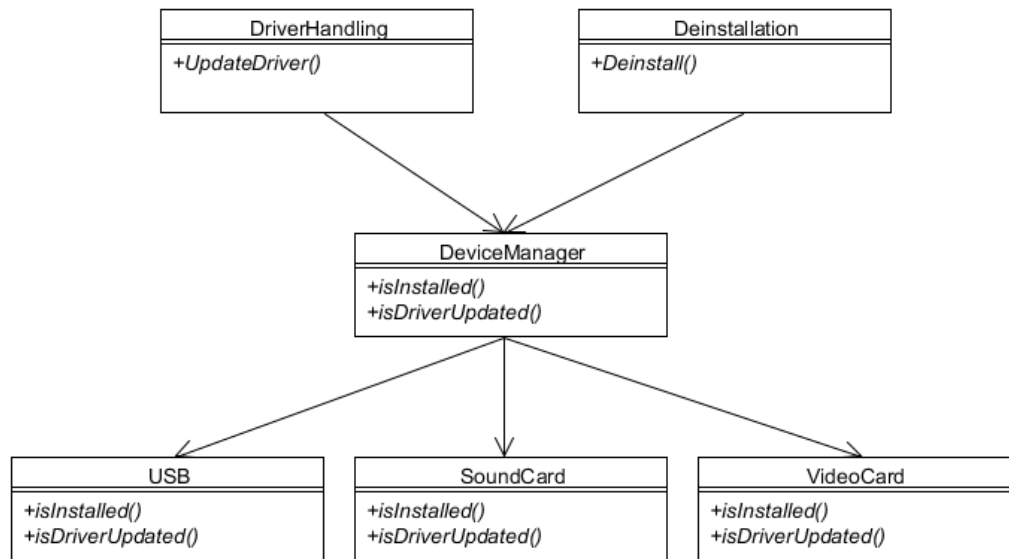


Figure 1.5.: Implementation with a Manager class cf. (Reddy, 2011, p. 59)

Encapsulating Functionality Separately from an Interface

For a recap: it is good practice for decreasing the coupling between software components. If a class uses functionality of an interface, it depends of some implementation of it. As more functionality a specific interface provides, as more those functionality would be used by clients. If they recognize a default implementation of their need, then they will use it instead of developing their own implementation. This leads to an increasing coupling. For that purpose, it is recommended to encapsulate functionality of an interface, e.g.: if a new functionality has to be added. The following simple C# interface in Figure 1.9 of a bank account implementation should point it out.

Listing 1.9: Encapsulate functionality if it is possible

```
public interface IBankAccount
{
    void AddAmount(int a);
    int Balance { get; set; }
}
```

If another functionality, e.g.: a method for determining the percentage rate of the specific account (assumption: percentage rate depends on the balance) has to be implemented, then a dilemma will be faced, because everyone who uses this interface has to implement the new functionality too. Instead of that, a new class *CalculatePaymentOfInterestOfAccount* could be implemented, which takes an instance of a *IBankAccount*, which implements that interface for determining the payment of interest. The benefit from that is, that *CalculatePaymentOfInterestOfAccount* is reusable and works with different implementations of the *IBankAccount* interface (Jenkov, n.d.).

1.2.7. Documentation and Testing

Providing a good documentation of an API is inescapable, because without the knowledge for the purpose- it becomes very hard to reuse something. In the best case, every class, constructor, interface, method, parameter or exception is described in detail. A class should be described in that way, that

1. API Management

everybody gets an idea of what a concrete instance of that class represents. An explanation of a method has to point out the special contract to the customer. Preconditions, postconditions and possible side effects have to be mentioned too. The documentation is finished, if there exists really no room for any different interpretations (Bloch, 2007).

It is a very bad practice, if the documentation is created after the finished implementation of the API. Another aspect is the person who writes the documentation: this person, should not be the one, who has participated in the development of the API (Henning, 2009).

According to keep the quality high- tests have to be provided. It has to be ensued that by adding a new feature, no existing use cases will not work any more. Reaching this level of assurance is only possible by intensive testing. See [Testing](#) for more details (Reddy, 2011, p. 62-63).

1.3. Practical Guidelines for Building High Quality APIs

This chapter concerns about patterns for the development of well designed APIs.

1.3.1. Singleton Pattern

General Description and Common Implementation

1. It is ensured that only one object of the class will be created
2. The instance can be accessed globally

Those two characteristics are assured by implementing the Singleton Pattern. (Townsend, 2002)

This pattern should be used for modeling objects, which exist singular, e.g.: for a logging, a scheduler for printer jobs or the already mentioned Manager Class. As mentioned a Singleton class can be used instead of a

1.3. Practical Guidelines for Building High Quality APIs

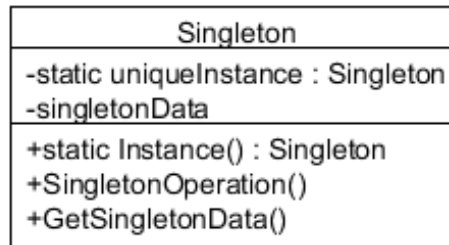


Figure 1.6.: UML representation of a Singleton class cf. (Townsend, 2002)

global variable, if more manageable options are necessary.

An example for an implementation in C++ is shown in Figure C.15. By just calling *GetInstance()*, an user gets access to the Singleton and it is only allocated if this command is used. It is recommended to set among others the constructor and destructor operator to private- in that way it is not possible for deriving classes from the Singleton. If it should be allowed, it can be declared as protected. The initialization of the Singleton can be performed, by using a local static variable, as seen in Figure C.16 (Reddy, 2011, p. 76-79).

Using Dependency Injection for Singletons

As seen in Figure C.17, the class *ExampleClass* depends on *Logger*, because by creating an instance the constructor of *Logger* is called. If the constructor of *Logger* changes the calling convention, then *ExampleClass* has to be adapted too. For avoiding that scenario, a technique called "dependency injection" can be applied. The concept is to pass an instance to class, by injecting it. An implementation, which should be avoided and seen in the code snippet, is to create the class and let the instance be responsible for the storing. Another aspect, which should be considered, is the performance of this implementation, because every instance of *ExampleClass* will allocate its own instance of a *Logger*. A recommend solution, as shown in Figure C.18, would be to pass a already created instance of *Logger* to *ExampleClass*.

1. API Management

The advantage of this kind of implementation with using dependency injection, is that the *ExampleClass* does not have to know anything about the parameters of *Logger* (Reddy, 2011, p. 81-82).

1.3.2. Factory Pattern

General Description and Common Implementation

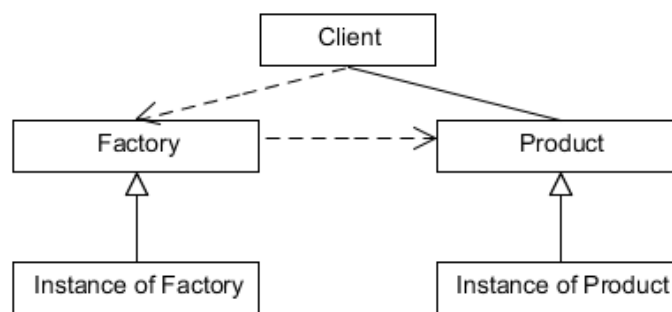


Figure 1.7.: UML representation of a Factory pattern cf. (Purdy, 2002)

This pattern is a widely used one. By using an instance of *Factory*, an instance of *Product* can be generated, as seen in 1.7. A client does not need to know anything about the different subclasses (Purdy, 2002).

It is common for using this approach for inheritance, in that way, that a derived class is able to override the method of the base class for returning an instance of the subclass. In Figure C.19, the method call *GenerateFurniture* will return a specific instance of derived classes, of course not of the type *Furniture* because this is the abstract base class. Figure C.20 shows the implementation of three concrete classes (desk, chair and cupboard). Following becomes evident:

1. The API encapsulates the derived classes
2. Users can decide at runtime, which specific concrete class should be created

1.3. Practical Guidelines for Building High Quality APIs

3. Headers of the derived classes are exposed only in "FurnitureFactory.cpp" and not in the public corresponding header "FurnitureFactory.h"
4. No client is able to see the implementation details of the different subclasses

Those benefits will be received by using a factory pattern ([Reddy, 2011](#), p. 85-88).

In general, a factory method should be preferred in contrast to a constructor, because it enables the API to be more flexible. There is no need to create an object of the type of class, for which the constructor was implemented, instead of that an instance of a subclass can be allocated. Using polymorphism may improve the code structure of the API. For that reasons, preferring a factory pattern to a constructor would be a good recommendation ([Tulach, 2008](#), p.72).

1.3.3. Proxy Pattern

Proxy and adapter pattern, can be called "Wrapped Patterns". Usually, it is a common practice to implement a wrapper interface, which is placed on the top of classes. Instead of defining a new architecture for a software, a wrapper interface for a cleaner API can be implemented. A negative influence is the decreasing performance, cause of redirecting by the wrapper interface.

By implementing a proxy design pattern [1.8](#), a one-to-one forwarding interface to the target code is realized. The call of *FunctionOne()* in the proxy class implies a call of the corresponding function (*FunctionOne()*) in the real object class. It is obvious that the proxy and the real object class have the equivalent interface. Generally the concrete implementation consists of a proxy class, which holds a reference to the real object class, so that a call of the proxy methods can directly be redirected to the object methods. A drawback of that would be to accept some code duplication.

Figure [C.21](#) shows a simple example of a proxy implementation in C++. By setting the copy constructor and the assignment operator as private, it is

1. API Management

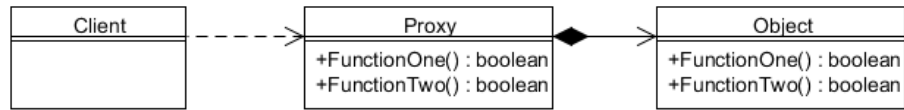


Figure 1.8.: UML representation of a Proxy Pattern cf. (Reddy, 2011, p. 91)

prevented that clients can copy from the object. This specific pattern can be used if the interface of the real object class must not be changed, but if the behavior should be adapted or if the real object class is a third party library, so changes are not possible to realize.

The benefits of using a Proxy Pattern are:

1. An instance of the real object is only allocated if there is a method call
2. The implementation of a permission layer between proxy and real object class is possible (for instance if some methods must not be called by clients)
3. Insertion of some logging statements for logging all references to the real object are easy to implement.

For that, it is a well known pattern too (Reddy, 2011, p. 91-94).

1.3.4. Adapter Pattern

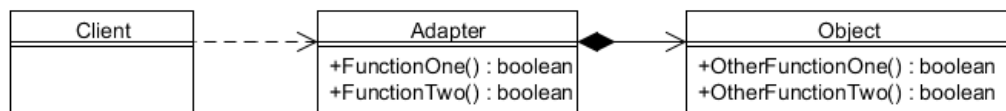


Figure 1.9.: UML representation of an Adapter Pattern cf. (Reddy, 2011, p. 94)

1.4. Maintenance of Backward Compatibility

The Adapter Pattern, as seen in UML representation in Figure 1.9, is used for providing a different interface for a class, but which is still compatible. Like the Proxy Pattern, it is a single component wrapper too. Usually this technique is suitable if a different interface for an existing API is implemented to make it work with another code, as shown in Figure C.22. What should be noticed in this example: the method of *SquareAdapter* has a different calling convention than the method in *Square*, but the functionality is still the same. Adapter patterns could be implemented with the usage of composition or with inheritance. In that case *SquareAdapter* would be a subclass of *Square*.

The use of this pattern receives following benefits according to API design:

1. **Improvement of Consistency:** Multiple classes with different interfaces could be equipped with a consistent interface.
2. **Hiding a Dependent Library:** A third party library for instance could be masked by the API. A client will not recognize a call of this third party library.
3. **Different Calling Convention:** An API which is implemented in C, could be provided as e.g.: C++ version by wrapping the C call into C++ classes.

The implementation of an adapter pattern is a popular practice too (Reddy, 2011, p. 94-96).

Therefore, this kind of pattern can be used if developers need to integrate and to work with a class, which is not compatible to the interface (Sugrue, n.d.).

1.4. Maintenance of Backward Compatibility

Some rules have to be considered if an API is adapted in some way for a new release. This chapter should give practical advices for keeping the backward compatibility.

1. API Management

1.4.1. Adding Functionality

Adding functionality, by implementing new classes, new methods or non-member functions (free functions) is rather unproblematic in the context of source compatibility because the new API has just to be a superset of the old one.

Of course, there are exceptions to that rule: adding a new pure virtual function to an abstract class will definitely break the backward compatibility. By considering the C++ example below, the method *NewPureVirtualMethod()* will be implemented for the new release. This leads to a problem, because every derived class has to provide an implementation for that method. The client code can not use the new version of the API without adjusting the code.

```
class SampleAbstractClass
{
public:
virtual ~SampleAbstractClass();
virtual void ExistingPureVirtualMethod() = 0;
virtual void NewPureVirtualMethod() = 0; // for the new release
};
```

A better solution would be to add new virtual functions to the abstract base class, instead of pure virtual functions (Reddy, 2011, p. 256-257).

```
class SampleAbstractClass
{
public:
virtual ~SampleAbstractClass();
virtual void ExistingPureVirtualMethod() = 0;
virtual void NewPureVirtualMethod(); // for the new release
};
```

Adding a static method may cause a source compatibility break, if an overloaded variant of this method is already implemented, e.g.: in derived subclasses. It will not be a problem according to binary compatibility (Tulach, 2008, p. 90).

1.4. Maintenance of Backward Compatibility

According to classes or interfaces: adding functionality by providing a new class or interface will not break binary compatibility, but there exists risks for source compatibility. Source compatibility can be violated by providing wild card imports (Tulach, 2008, p. 89).

1.4.2. Changing Functionality

There is a big difference according to valid changes between source and binary compatibility:

In the context of source compatibility, it is possible to add optional parameters after previous implemented ones, as shown in the the next C++ code. The compatibility is maintained by switching from 1.0 to 1.1, because there is no need to pass a second argument in 1.1 (Reddy, 2011, p. 257-258).

```
// version 1.0
void RandomMethod(string requiredParameter)
// version 1.1
void RandomMethod(string requiredParameter,
string optionalParamter = "default_value")
```

Another possibility would be to implement a new method, which simply calls the old one, but that should not be a desired solution. The reason is that it could be a risk for adding a new methods into classes, which will be probably derived (Tulach, 2008, p. 96-97).

Changing the return type is valid too, if the previous return type was a "void". This works because the client code will not check the return value-based on the fact that this will not happen in version 1.0. Such a change is shown in the C++ example below.

```
// version 1.0
void SetSomeString(string firstParameter);
// version 1.1
bool SetSomeString(string firstParameter);
```

1. API Management

Maintaining binary compatibility is much more difficult, because any change of the signature will cause a break. If it is mandatory for maintaining binary compatibility and to change the signature of the function, then a new method has to be created, which will overload the name of the old function.

```
// version 1.0
void RandomMethod(string firstParameter);
// version 1.1
void RandomMethod(string firstParameter);
void RandomMethod(string firstParameter, string addedParameter);
```

By just fixing bugs in the implementation of functions, neither source or binary compatibility will be broken- but of course the functional compatibility. Most of the clients will support those adaptations, but some may not. For this purpose additional functionality can be delivered as an optional feature in that way that a client is able to turn it on and off ([Reddy, 2011](#), p. 258-259).

1.4.3. Deprecating Functionality

A feature will be called deprecated if it will be removed in the next release, but it still exists in the current version- so clients can call it. Of course they should be noticed by a warning, that this specific feature will not be supported in the future and a recommend feature should be provided instead of the deprecated one. Deprecating functionality is a way for providing clients a deadline while they can switch to an alternative feature ([Reddy, 2011](#), p. 259-260).

So, a good advice according to things, that should be done for deprecating functionality is:

- Making a notice in the API documentation to inform clients and developers about it
- Releasing of a minor version with the deprecation, that users will have time to switch to the new functionality

The API developers have to give the clients some extra time for switching ([Preston-Werner, n.d.](#)).

1.4.4. Removing Functionality

This is the final step after marking a feature as deprecated. Removing functionality will of course cause a break of client code which is using that specific feature. For that purpose it should be marked as deprecated in the previous version of the API. Some companies provide older APIs to their clients, so that they can use the old functionality if they do not want to switch ([Reddy, 2011](#), p. 261).

To be more specific: removing a method will lead to compatibility problems (provided that this specific function will be used), because this will violate source compatibility. Code, which calls this function will not compile any more. There is a problem too according to binary compatibility, because if a code will be compiled against an older version of the API which contains the function and then executed against a the current version, then a runtime exception will occur ([Tulach, 2008](#), p. 88-89).

If a member of a class or an interface is referenced or its methods are called, then this class or interface is definitely used. Removing those kind of classes or interfaces will break source and binary compatibility ([Tulach, 2008](#), p. 89).

1.4.5. Versioning of APIs

Once the initial state (version 1.0) of an API is reached, but the development of an API is of course not finished. Many changes and improvements (bugs, new features) have to be done probably in the future and in the best case no client will recognize any change when releasing a version of the API. In this case the process of API management seems to be stable. If clients have to adapt their code because of changes in the interface, they will get annoyed. So for that, an API with a reputation for stability and high quality could be a serious factor with regard to the success of a product ([Reddy, 2011](#), p. 241).

1. API Management

Version Numbers

Providing a number for every release of an API is a common way for distinguishing between the different versions.

There exists many different schemes for providing a versioning scheme- one established one is to provide three numbers for each version. Following figure 1.10 should help to explain the different meanings of the numbers according to a "Major-Minor-Patch" scheme:

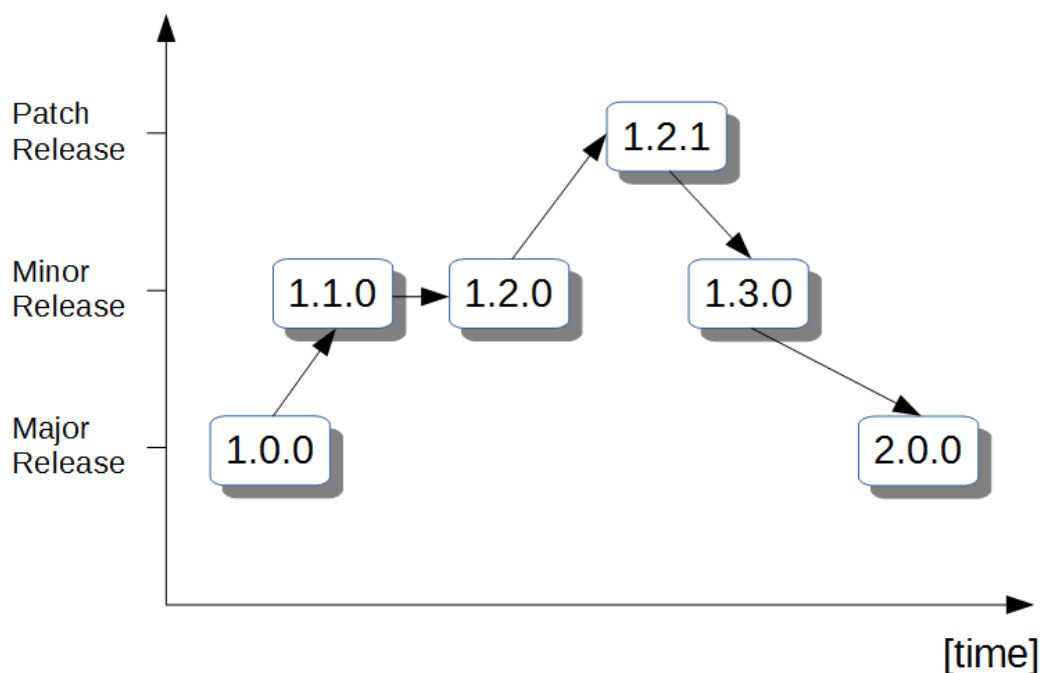


Figure 1.10.: Example of a "Major-Minor-Patch" numbering scheme cf. (Reddy, 2011, p. 242)

1. **Major Number:** Determines the first integer in a version number- in general it starts with a "1" for the announcement of the initial state. This number will be only increased if important changes are made. According to API versioning, this special number can give a reference to the backward compatibility.

1.4. Maintenance of Backward Compatibility

2. **Minor Number:** This number is set to zero when releasing a new major version. An increase will be executed if small changes (e.g.: new small features or important bug fixes) are implemented. By changing the minor numbers the compatibility between the different versions of the API should always be guaranteed. If some new features will be published, then it would be recommend for not switching to an older version if adaptations of the software are not allowed.
3. **Patch Number:** After releasing a minor version, this optional number is set to 0. It will be increased if important bugs or security relevant issues are fixed. According to API management, different patch versions should maintain forward and backward compatibility: for that a client can revert to an older version and than again switch to a newer version without need for making changes in the code.

Further possibilities:

- **Build Number:** An additional number according to this scheme would be for example an automated build number for distinguishing each build of the software.
- **Symbol:** In many cases a symbol is added to the versioning string, which indicates the phase of the development. For instance an "a" for the alpha-, a "b" for the beta release and a "rc" for the release candidate.

So much on the topic of version numbering ([Reddy, 2011](#), p. 241-243).

One advantage of the "Major-Minor-Patch" numbering scheme- comparing the above possibilities, is that it specifies some rules according to simple bug fixes or incompatible API changes. This is very important for dependency management, because without some specifications, version numbers may defeat the purpose ([Preston-Werner, n.d.](#)).

Stages of an API Lifetime Cycle

The life cycle of an API consists of four different phases, which will be described in this section. An important aspect is the big difference between the maintenance of an API and an ordinary software product: A change in the API has a more important effect, because of the influence according to

1. API Management

the clients. A simple practical example would be a change of the signature of a method: there are no further consequences in a simple software product but of course in an API if this method is used by clients or colleagues. The most critical transition is the time after the initial release, because after this point the rule has to be observed, which was contracted to the clients for providing backward compatibility. The time before the initial release is the last chance to make major changes.

- **Pre-Release:** In this phase, an API goes through the default software cycle: Plan, Design, Implementation, Test. It is a good practice to provide an early API version to the clients, for getting feedback. As already mentioned, this is the last chance for a big redesign and for significant changes. This version is indicated with a number like "0.x". It has to be considered, that this API may change significantly in future.
- **Maintaining the API:** Modifications are still possible, but they have to be restricted to adding new classes and methods only. Now the API contract, has to be observed permanently. The usage of API reviews and tests should avoid incompatibility.
- **Completion:** At some time, an API may reach maturity- so no changes will be made in future. This can be true if the API solves exactly the problem it was made for or that the developers will switch to another project. Bug fixing will still be necessary.
- **Deprecation:** Some APIs will be deprecated, e.g.: if they do not provide a useful service any more. It should be prevented using deprecated APIs for new development.

Those are important aspects about the life cycle of an API (Reddy, 2011, p. 249-250).

1.4.6. Branching

Especially bigger software projects use a specific branching strategy, for simultaneous development, for providing release and maintenance branches. In the following chapter, different aspects of a branching policy will be treated (Reddy, 2011, p. 245).

1.4. Maintenance of Backward Compatibility

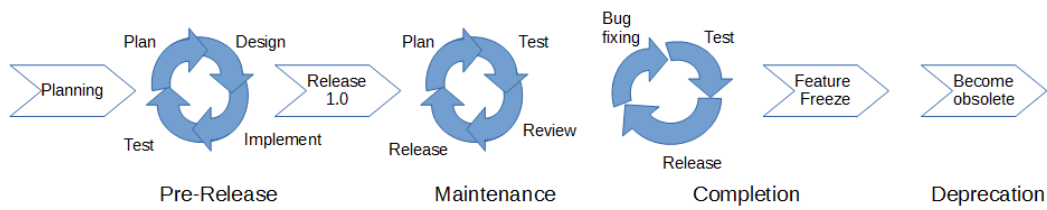


Figure 1.11.: Life cycle of an API cf. (Reddy, 2011, p. 249)

Specific Branching Strategies

Every strategy consists of a "trunk" code line, which is used as main branch and has always to be stable. Subbranches are created from this main branch, for the development of new features or for specific releases. In this model, parallel development is possible: while creating new features, releases can close some changes to give the existing project more stability.

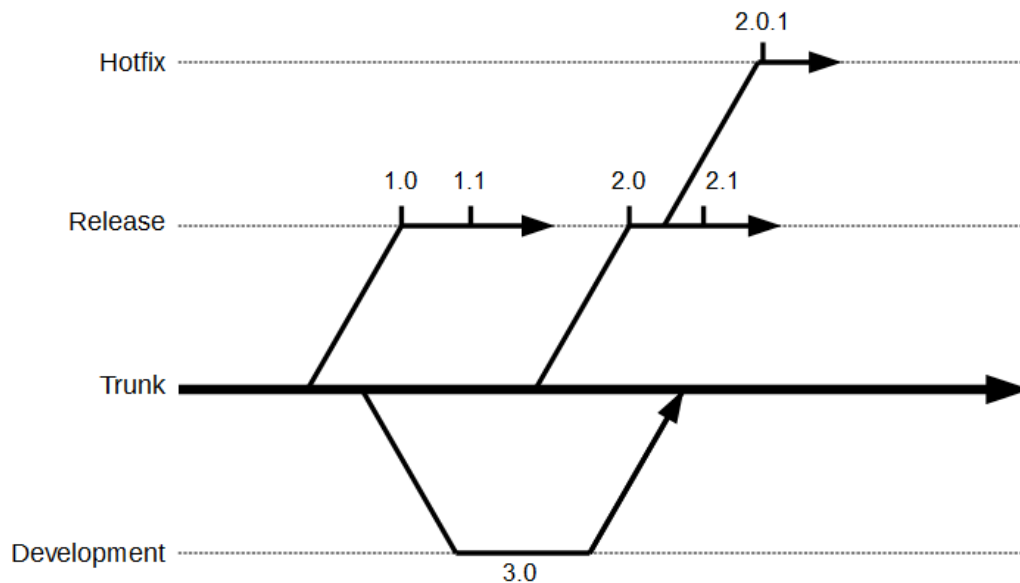


Figure 1.12.: Example of a branching model cf. (Reddy, 2011, p. 246)

1. API Management

Figure 1.12 shows a popular branching model: every major release gets its own branch and along those lines, minor releases will be made. For making a so called "hotfix", a new branch will be created from this line. The line at the lower end illustrates a development branch. Such a strategy receives the benefit that it is possible to make longer term development without skipping or postponing a release (Reddy, 2011, p. 246).

Branching Policies

It is a common practice for creating multiple branches for parallel development and release management. There are several policy decisions to finalize, e.g.: how many development branches are allowed, at which time should release branches created, what is the merging strategy between different branches, ect. Longer term development should be realized with corresponding development branches and it is important that developers and QA engineers are working on the trunk code line too and not only on release branches for keeping this branch stable. Different branches should be merged frequently- to keep the code divergence low. The choice of the source control management system has a big influence too: tools like "Git" or "Mercurial" are predestined for making parallel development with different branches possible (Reddy, 2011, p. 246-247).

APIs and Branching

Evolving and maintaining APIs with multiple different branches with multiple teams could become a challenge. Generally in larger projects there are release branches and development branches. How can it be ensured, to provide a consistent process for API development? The following few guidelines should help for improving this process according for using multiple branches (Reddy, 2011, p. 247).

Avoiding Unsupervised API Changes in Subbranches

Considering a situation in which the software project consists of different release and development branches: One of those release branches needs some API changes and the developers are going to do their work. This

1.4. Maintenance of Backward Compatibility

specific changes may get lost if they will never be merged down in the trunk branch. For avoiding this situation, changes should be committed to the trunk and then merged to the release branch (Reddy, 2011, p. 247).

Keep the Trunk Branch Up to Date

Changes to a public API should either be made in the trunk branch or the specific branch is merged into the trunk as soon as possible. It is a good practice to synchronize the different development branches frequently with the trunk line. This practice prevents besides programming conflicts of different teams with their corresponding branches (Reddy, 2011, p. 248).

Making API Reviews

An API review (see [API Reviews](#) committee should be installed for proving the changes in the API and especially whether there exists a problem according to the backward compatibility. All members of the project, of the team imposes a burden to this committee, because they are the last line of defense. They determine whether this important changes are ready for release or not (Reddy, 2011, p. 248).

1.4.7. API Reviews

The company NetBeans has made many experiences with API design: they tried to deploy one single API architect, but this person became a bottleneck. The effort was overwhelming, so they switched to a mode in which responsibilities were distributed to a group with one moderate dictator. It has exposed, that none of those is perfect for API design. One important aspect, which the NetBeans team discovered, is that APIs are very important for the communication amongst users. They tried to deploy a open process according to API development: Everybody, who has some reference to an API, can make a request according to a change. After that, a group of people will make a review for the proposed changes. An API has a huge influence to many users, so every developer is an API writer too at NetBeans (Tulach, 2008, p. 54).

1. API Management

To be more specific, Martin Reddy advises to provide meetings before the API is released, a so called "Prerelease API Review". At least following persons should be present at this review:

- **Product Owner:** Represents on the one hand the wishes of the clients and on the other hand he is responsible for the product planning.
- **Technical Lead:** Has a very good knowledge about the code and is able to argue why specific changes were made.
- **Documentation Lead:** A well done documentation is essential for the API development process. For that, a good technical writer should be present.

So much on which people should be part of this "Prerelease API review" (Reddy, 2011, p. 263).

1.5. Testing

It is probably unavoidable that bugs will be implemented by developers- it does not matter how experienced they are. The growing of an API will also increase the error rate. So it is obvious to test the API. Testing is a critical aspect in the whole API development process, because by writing tests the quality and stability of the interface will be proved. Without a reliable product, clients will necessarily move to a competitor. It also ensures that there will not be a break of the clients code, given that the tests are well implemented (Reddy, 2011, p. 291). A developer, who will never forget to do continuous testing, will probably more efficient than someone who does not. That is because the earlier the testing start, as sooner improvements for the software will occur (Tulach, 2008, p. 150).

1.5.1. Benefits of Implementing Tests

Many developers do not like to write tests but probably no one will denote testing as a waste of time. Tests should be written continuous and as soon as possible, because the costs will grow as long a bug will not be detected, a

1.5. Testing

good example therefore is the waterfall development process. Determining a failure at the end of the development process will face the team with a difficult task, because there is probably not much time left. It is a good advice, that the management of the project will explicitly declare tests as an requirement and also schedule time for that. There exists many reasons why tests should be written (Reddy, 2011, p. 291-292).

- **Confidence about Quality:** Automated tests will give the team confidence about the quality of the code. It is comforting too, if all tests pass after implementing a change. In other words: providing automatic tests will increase the incentive for improving the software.
- **Backward Compatibility:** Tests are suitable for capturing the behavior of old versions of the API- because it is mandatory to maintain the backward compatibility. Software, which was written against an earlier version of the API has the work with a newer version too.
- **Saving Costs:** Detecting an error in the API late in the development process will be very expensive. Fixing an error as soon as possible is a recommend advice.
- **Use Cases:** Use cases, which can be executed by customer, can be represented by tests. If those tests succeed, then the required functionality will be reached.

As it can be seen, developers who are implementing tests, will receive a lot of benefits (Reddy, 2011, p. 292-293).

1.5.2. Disadvantages by Writing a Huge Number of Tests

An implemented test suite needs of course maintenance. By increasing the amount of tests, the effort for that will get higher too. In some situations, a valid change of code which will improve the software, can lead to the failure of many tests. In that case, the developer has to fix all failed tests, which can take many hours. It is also a very frustrating task, because the developer who has done high quality work, has now to handle many tests. In the further sections, methods will be explained for avoiding those specific situations (Reddy, 2011, p. 293).

1. API Management

1.5.3. Categorization of API testing

In general, software testing can be divided in:

1. **White Box Testing:** Achieved with a programming language, with knowledge about the code.
2. **Black Box Testing:** Achieved without knowledge of the concrete implementation. This approach is based on product specifications.
3. **Gray Box Testing:** Includes aspects of white box and black box testing: Black box testing is applied by exposing the source code.

Those testing types can be applied to both: API testing and end user application testing. The main difference is that API testing can only be performed by writing code against the API, so many common testing strategies will drop out, e.g. system testing because it is essential to have a complete integrated system for that. GUI testing is not practicable too for API testing, because there simply exists no graphical user interface. The most efficient testing approach for APIs are tests, which are written in code and can be fully automated- so the main focus will be on unit testing and integration testing.

- **Unit Testing:** For testing the smallest functionality of code and ensuring that the code fulfills the expectations of the developer.
- **Integration Testing:** Comprises diverse functionalities of the software and ensures the expectations of the user.

Apart from this, there exists a wide range of non functional tests:

- **Performance Testing:** Refers to a required speed and memory usage of the API functionality.
- **Load Testing:** Is the API persistent against some demand or stress?
- **Scalability Testing:** Proves whether the API is designed for large and complex data too.
- **Soak Testing:** Can the software be applied during a long period of time?
- **Security Testing:** Does the software fulfill confidentiality, integrity and authentication?

- **Concurrency Testing:** Is the software suitable for multiple number of threads?

As mentioned, those tests do not have any relation to the functionality of a software (Reddy, 2011, p. 293-294).

Unit Testing

This testing approach ensures that the smallest piece of functionality (e.g.: a single method or a class) of the software meets the requirements. This specific functionality of the API will be considered in isolation. Those tests can be implemented by developers, who have a knowledge about the corresponding code. So this testing approach belongs to white box testing. In many cases, the method which should be tested, refers to others objects or external resources (database, network connection). For that, unit testing can be applied by a fixture setup or with help of stub or mock objects.

- **Fixture Setup:** Is used for creating an environment before each unit test is applied. E.g.: initialization of singletons, preparing dependent files, adding data sets to a database. In general a method called *setUp()* is responsible for that. For a clean up of the objects, after the different test runs are finished, a method named *tearDown()* is used.
- **Stub and Mock Objects:** The code is tested in isolation, but to be able to refer to some dependent objects, like a database, a stub or a mock object is created for that purpose. If a query to a database should be sent, than a stub database object can be implemented, which accepts some queries and makes a response without connecting to the real database. The advantage of that achievement is, that errors according to a database connection, or network problems can be excluded. The main difference between a stub and a mock objects is, that a stub object is created for a specific set of unit tests, while the mock object is more adaptable.

So, unit testing is not as trivial as it may sound (Reddy, 2011, p. 295-297).

1. API Management

Integration Testing

An integration test considers the collaboration with different components, which should separately be proved by unit testing. Just unit testing is not sufficient because this does not guarantee that the different components are in harmony together. E.g.: the interface of one component is not compatible with another component. Integration testing belongs to black box testing, because they should represent the usage of clients (Reddy, 2011, p. 297-298).

Performance Testing

An API needs an appropriate level of performance, this depends on the purpose of the application: if the API is used for a complex game engine, than this aspect would take an important role. The corresponding performance test fails if a defined threshold will be outvalued. If that happens, the code has to be optimized. By writing those kind of tests, it is ensured that changes in the API will not have any influence on the performance. Implementing and maintaining performance tests is more difficult than doing the same with integration tests, because those tests depend on the underlying hardware. This implies different thresholds according to each used machine. One attempt would be to run performance tests on multiple machines and storing the results in a database. This will lead to the collection of a huge amount of data and faces the developer with a data mining problem. The developer will have to spend much time in examining the test results. For that, the top 5 or 10 most hardest changes should be observed (Reddy, 2011, p. 298-301).

1.5.4. How to Write Good Tests

What makes a test good and efficient and which common techniques exists for that? Those aspects will be covered in this sub chapter.

Qualities of Tests

Tests should be developed with the same precision as the corresponding API code. By keeping the following attributes as high as possible, it is possible to implement and maintain a stable test suite, which is important for improving the quality of an API.

1. **Fast:** This attribute is important for getting fast feedback about the test results. In general, unit tests run very fast and often take just a couple of seconds for each test. In contrast, an integration test will need longer to deliver results. It would be a good advice to divide tests in "fast" (checkin) tests, which run after every build and "slow" (acceptance) tests, which will be started before a release.
2. **Stable:** A test should deliver the same result, every time the test is started. If this does not work, using a mock object may be a solution.
3. **Portable:** An API which is compatible on multiple platforms, should be testable on the same platforms too.
4. **Coding Standard:** The core statement is to keep the same conditions for the API development to the tests. E.g.: if reviews are hold about the API, then the same should be applied for testing too. It is also mandatory to document tests and giving indication about a failure.
5. **Reproducible Failure:** Providing as much logging information as possible, for tagging the concrete point of failure.

Ideally, every of those qualities will be reached with a sufficient degree (Reddy, 2011, p. 301-302).

Testing Strategy

As mentioned, there is a difference between an unit test and an integration test, because of the fact that the source is known by creating an unit test. In general, the aim of both is to prove the functionality of the API methodically, this can be done with following techniques:

- **Proving Conditions:** By implementing unit tests, it is necessary to cover all possibilities according to *for*, *if-else*, *while* and *switch* statements.

1. API Management

- **Determining Equivalence Classes:** If there exists a function, which accepts only numbers with a range from 0 to 10, than there are three equivalence classes (negative numbers, numbers from 0 to 10 and numbers greater 10), which should be covered by at least one representative.
- **Prove Boundaries:** There are many errors because of boundary violations: if an element is stored in e.g.: an array with length n , that it should be tested to store the element at indexes $0, 1, n$ and $n-1$.
- **Parameter Combinations:** All possible combinations of parameter, which can be passed, have to be tested.
- **Returning Value Testing:** The return values have to be proven. This can be a simple return value of a function or a parameter, which is passed as reference or pointer, and includes the result of the operation.
- **Getter/Setter Testing:** Calling the getter method before the corresponding setter should return the correct default value.
- **Proving Backward Compatibility:** Tests have to be provided, which ensure that the backward compatibility is given after adapting the API.
- **Negative Testing:** Invalid operations have to be achieved for determining how the API reacts. E.g.: the operations of the API need a working ethernet connection, but what will happen if the connection is retired?
- **Buffer Overflows:** This can be a very critical part, because a buffer overflow may overwrite some security critical variables. It has to be proven that a buffer overflow can not occur.
- **NULL Pointer:** If a pointer can be passed, tests should be provided which prove the behavior when they hand over a NULL pointer.

By applying this, a higher quality should be achieved for testing ([Reddy, 2011](#), p. 302-304).

1.6. Conclusion

This chapter gave an overview about the concept of an API and the aspects to consider for achieving a high quality for it. Design patterns, best practices according to the implementation and testing strategies should support

1.6. Conclusion

this undertaking. Besides having experience in developing APIs, foresights according to extendability and maintaining backward compatibility of an API are seen as best practices. By considering all this technical components, it should not be forgotten what the main purpose of an API should be: an API should solve a specific problem of a customer and applying it should be as easy as possible.

2. Agile Software Development

In this chapter, the agile development approach will be explained with a brief view on traditional approaches. At the end, an observation is made on how the agile approach differs from the traditional one. The purpose of this chapter with regard to the whole work is to improve the knowledge about agile software development and to benefit from synergy effects with the previous chapter. The first chapter deals with API management and those practices and guidelines should be applied in the context of agile approaches. On this account, it is necessary to find out what agile software development is about. E.g.: how can specific practices of the previous chapter be applied to agile approaches, instead of classic approaches?

2.1. What is Agile Development?

In software development, the term "agile" is inseparably linked to the following principles: collaboration, flexibility, transparency, simplicity and responsiveness. Any approach of project management, which requires unite teams is based on those principles within this context. A better knowledge about agile software development can be reached by inspecting the contrary approach: the traditional software development ([Huston, n.d.](#)).

The most popular traditional approach is called "Waterfall model", which will be called into question in this work.

2. Agile Software Development

2.1.1. In Contrast to Agile Development: the Waterfall Model

Besides the "Agile" model, the "Waterfall" model is one of the most popular approaches. Both are not new and the waterfall model is not much older than the agile approach, although it gives the impression it should be. The waterfall model, which is also named the "traditional" model, consists of different steps, which are processed sequentially. The next step can only start if the previous step is finished. Therefore, it is easy to understand. It starts by determining the requirements of the product, then customer and developer have to agree on a contract. Because of this, planning can be simplified and the progress of the project can be measured rather easy. There exists risks, that a customer will not be satisfied when delivering the product, because there may occur huge deviations between the planning ahead of the "real work" and the output. Another aspect is, that the customer can not clearly determine all details of the requirements of the product. The contrast between those two models is illustrated in Figure 2.1: The waterfall model is a sequential process: once one step is finished, the next begins- so after the "Analysis" follows the "Design". Unlike to the traditional approach, the agile one provides multiple steps, which consists of all phases (from Conception to Deployment) (Lotz, 2013).

Another example of a Waterfall process could look like this:

1. **Requirements**
2. **Design**
3. **Implementation**
4. **Verification**
5. **Maintenance**

It can be seen that "Design" and "Implementation" are previous and separate steps from "Verification" and "Maintenance". This is an aspect, which agile development wants to address. (Huston, n.d.)

2.1. What is Agile Development?

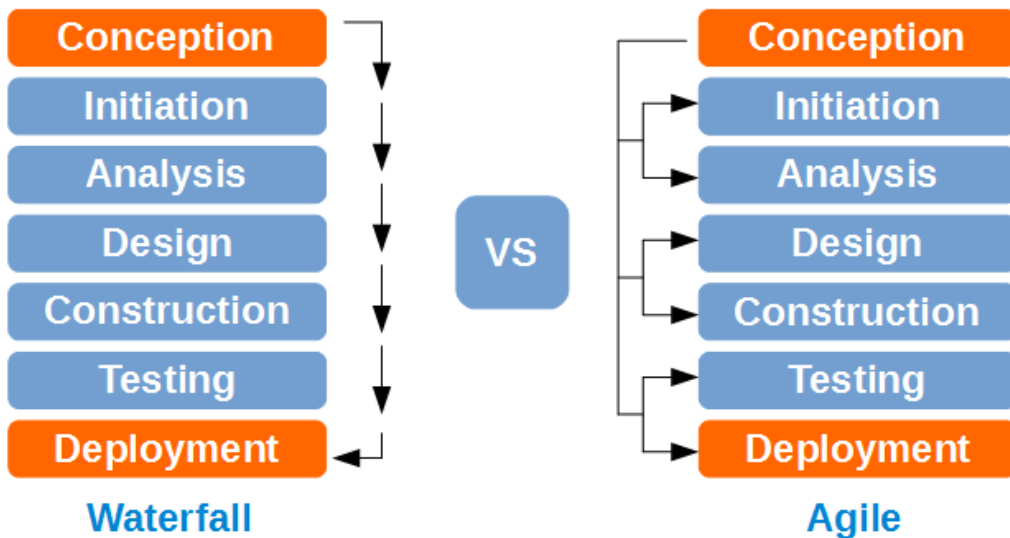


Figure 2.1.: Waterfall model vs. Agile model cf. (Lotz, 2013)

2.1.2. Key Aspects of Agile Development

Considering the problem of the gap between the creation of code and the testing leads to the conclusion that developers and testers have to improve their collaboration. They have to hold meetings frequently for exchanging their views. The agile way avoids that developers will refine a code until the customers are satisfied, without a simultaneous verification. Development and testing have to be merged- not in the context of people- but in context of time and processes. Agile development should motivate developers to think as testers and testers to think as developers. In general, all stakeholders of the development process, including customer, manager, developer and tester have to work closely together within a development process, which differs from the waterfall model. Another key term is "Iterative Development". Agile development is also based on feedback, which can be given from everyone, who is part of the implementation of the product. Feedback can be achieved through an iterative approach- therefore it is necessary to produce working versions of the product. Those working versions consist of course of just a part of all necessary features of the final product. As seen in

2. Agile Software Development

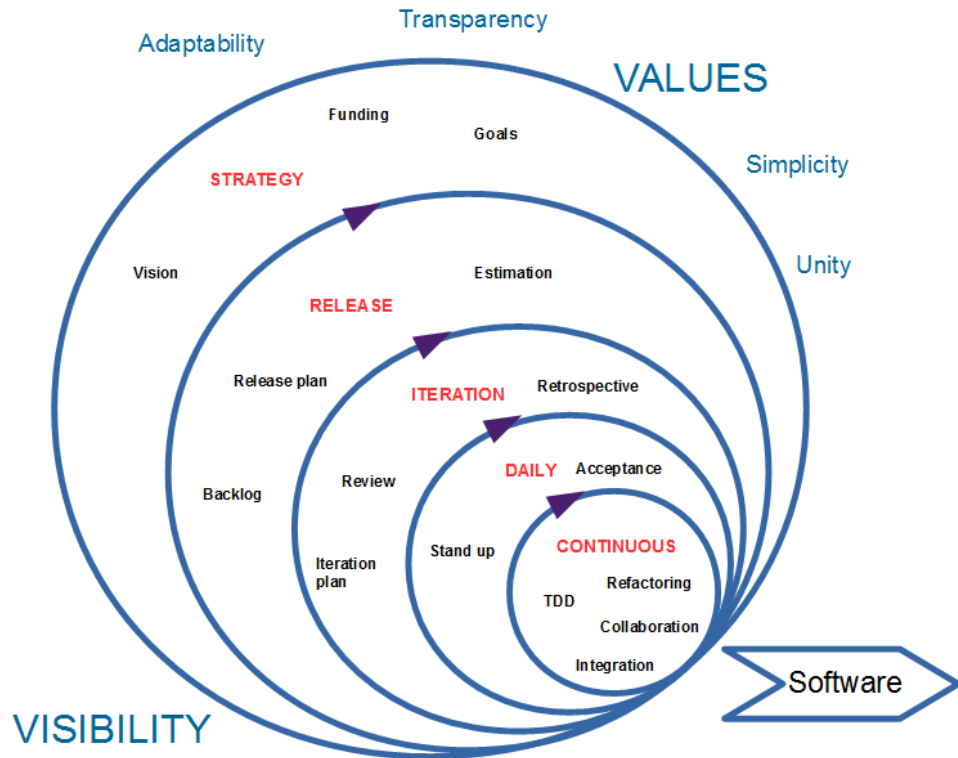


Figure 2.2.: Agile Development cf. (Huston, n.d.)

Figure 2.2, the agile development approach looks difficult and implementing it the first time will be a challenge, but if this is done and the whole team accepts it, then the company will receive a lot of benefits, see 2.3.2 (Huston, n.d.).

2.2. Agile Principles

There exist several agile principles, which can be applied in every agile approach and serve as guiding principles:

1. Customer satisfaction is the most important goal. This can be reached by delivering software early and continuous.
2. New or changed requirements have to be welcomed with open arms, because this is one competitive advantage of agile approaches.
3. Working software has to be delivered as early as possible.
4. Developers and customers should work together as a team and meet each other daily
5. Give motivated individuals confidence and build the projects around them.
6. Face-to-face conversation is the best way of communication through the project.
7. The best way for measuring the progress of the project is a working software.
8. The development has to be sustainable and the pace should be able to be maintained by the team
9. Strive for a good design and technical excellence.
10. Promote Simplicity.
11. Promote self-organized teams.
12. Teams will give feedback concerning how they can improve their work.

Those twelve principles are included in the Agile Manifesto ([Beck, 2001](#)).

2.3. Comparison of Agile and Traditional Software Development

This section deals with a comparison between agile and traditional software development.

2. Agile Software Development

2.3.1. TSDLC vs. ASDLC

To reach a deeper knowledge about the differences between agile and traditional practices, it is necessary to illustrate the Traditional Software Development Life Cycle and the Agile Software Development Life Cycle.

Traditional Software Development Life Cycle

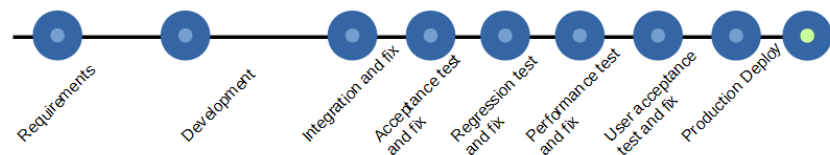


Figure 2.3.: Traditional Software Development Life Cycle cf. (Shoukath, 2012)

The *Traditional Software Development Life Cycle* is shown in Figure 2.3. So what are the properties of this life cycle?

At first, there is a requirement phase for determining the requirements until the development will start. A huge effort of time is needed, which cannot be used for other activities. If the requirements phase is finished, separate development work will start by different developers to perform the development of the product. After that, the work of the diverse developers has to be integrated, which can often cause rework. By finishing this phase, different testing phases have to be passed, including acceptance, regression, performance and user acceptance testing. If the testing has ended, the product can finally be deployed.

There are some interesting properties of this life cycle:

1. **Long Requirements Phase:** This phase, for determining the functionality of the product is quite long and delays the start of development- in context of the assumption that software will improve in quality the more detailed the requirements are determined.

2.3. Comparison of Agile and Traditional Software Development

2. **Late Use of Testing:** The testing phases are performed very late in the delivery chain. This may cause much rework if the tests fail. Another bad case would be if there is not enough time left for testing.
3. **Relation of Development to Deployment:** The time for deploying the product is very much longer in proportion to the development time.

Those properties allow to anticipate that there is a big scope for improvements (Shoukath, 2012).

Agile Software Development Life Cycle

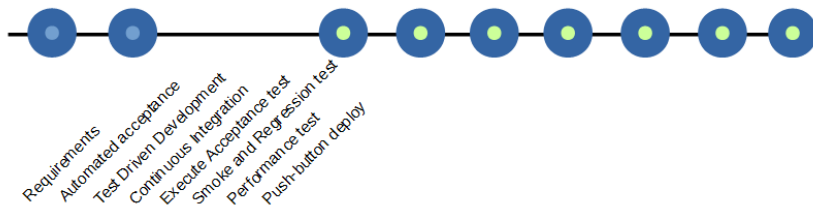


Figure 2.4.: Agile Software Development Life Cycle cf. (Shoukath, 2012)

The *Agile Software Development Life Cycle* approach differs from the traditional in many ways. At first, the long phase of determining the requirements will be avoided, the development starts without delay with a few basic requirements. The whole code has to be covered with unit tests and by performing Continuous Integration, there is no need for a separate Integration phase. The next phase, which will be avoided is the Acceptance test phase- because by using the Test Driven Development approach, automated acceptance tests will be provided and proofed by the Product Owner before the development starts. Smoke and Regression tests have to be automated too, so the cycle can eliminate the Regression test phase. The Performance test phase can be substituted by frequently running Performance tests. It is important to execute so many tests early in the life cycle and with a frequent communication and a close cooperation with the customer, the User Acceptance test phase should not be necessary any more. The deployment

2. Agile Software Development

itself can be realized with just a few small scripts. So the deploying the product is possible many times a day (Shoukath, 2012).

Another important aspect results from this approach: the customer has frequent insights into the development and can steer the project in the right direction (Lotz, 2013).

2.3.2. Advantages of Agile Practices

Transparency of the Development

Customers can determine the priority of different features and they are part of frequent reviews by the end of a development cycle. For that reason, they have a better feeling about the whole development work (Zolyak, 2013).

Flexibility

The customer has frequent insight into the product. For this reason, it is possible to react more quickly, if the customer is not satisfied about the current product (Lotz, 2013). The backlog items can be changed and new items can be added quickly, so they will be managed in the next development cycle and may already be achieved in just a few weeks (Zolyak, 2013).

Competitive Advantage by Focusing on Customers Needs

By the usage of user stories, the project team can generate a real need of the customer instead of realizing just a technical feature. Beta testing at the end of each development cycle allows to gain an early feedback, for improving the product in the next cycle (Zolyak, 2013).

2.3. Comparison of Agile and Traditional Software Development

Quality of Products

The main reason why products have a higher quality, is because of the fact that testing has to be part of the development process. Therefore, deviations can be detected earlier ([Waters, 2007](#)).

Collaboration with Customers

The agile way provides a close cooperation between the customer and the development team. Customers become a part of every step during the whole development, so it is easy for them to monitor the progress of the product. This improves the trust too, that the development team is able to produce a high quality software ([Zolyak, 2013](#)).

Enjoyment of Work

There is a greater enjoyment of work by living the agile way. Members of the team do not have to grapple with preparing a detailed specification of the product. For instance: workshops are more suitable for that. The whole team has a say concerning the requirements. This increases the motivation too ([Waters, 2007](#)).

Avoiding Rework and Improving Quality

Testing is integrated early in the whole process and the created tests will be run against the code frequently- this ensures the quality of the code and avoids bad surprises, which would occur at the end of the traditional life cycle ([Shoukath, 2012](#)).

2. Agile Software Development

2.3.3. Disadvantages of Agile Practices

Acceptance of Employees

A report determines that more than half (64 percent) of 200 participants are complaining about the hard switch to an Agile Development approach and forty percent stated that they have not received any benefit from it ([Smolaks, 2012](#)).

Danger of Reduced Documentation

Reduced documentation could be a possible risk of failing for some projects because detailed documentation is mandatory for future activities such as maintenance ([Smolaks, 2012](#)).

Hidden Costs

As development teams became smaller and deadlines occur more frequent, the average costs for projects increased while the possibility of software errors remained at a high level ([Smolaks, 2012](#)).

2.3.4. Advantages of Traditional Practices

Simplicity

Understanding this model is quite simple because it follows a linear procedure, therefore it is easy to implement it. Another benefit is, that documentation will be provided after every major step of software development ([Tilloo, 2014](#)).

2.3. Comparison of Agile and Traditional Software Development

Determining the Progress

As already mentioned, all requirements are specified ahead, therefore that the progress of the project can be detected rather easy (Lotz, 2013).

Simultaneous Working on Different Components

If multiple software components, which are depended on each other, have to be developed in parallel, then a concluded design phase could be a benefit (Lotz, 2013).

2.3.5. Disadvantages of Traditional Practices

Inflexibility

A big disadvantage is obvious: if one step is finished, then there is no way back. So a badly performed requirement phase will probably not lead to a successful finish of the development phase. Another weakness is that there are no plans to adapt new requirements of the customer after the Communication phase. It is likely, that a customer wants to refine his requirements during the whole process and this will lead to disorientation (Tiloo, 2014).

Late Recognition of Errors

Another huge problem is, that errors in the software are recognized late in the process and so adaptations require much effort (Tiloo, 2014).

Deployment

As seen in 2.3, there is a huge effort of time needed for the deployment. So, frequently deploying the product is not possible (Shoukath, 2012).

2. Agile Software Development

Time to Market

The long duration of deploying the product leads to a delay of the product in reaching the market (Shoukath, 2012).

2.4. Conclusion

Basically, agile approaches are geared towards handling the development with iterative cycles. The goal of each iteration is to provide a working software and demo it to the customer. There is no need to determine detailed requirements ahead of the development, basic functionalities are sufficient at the beginning because frequent adaptations will flow into the project, caused by a close cooperation with the customer. Feedback of customers is important, because the main goal is to satisfy them. Essential for this feedback is transparency, so that customer have insights into the development. This leads to a good cooperative work, which is an important factor for the success of the whole project.

3. Expert Interview

The previous chapters dealt with theoretical aspects of API management and agile practices. But how are those two big topics handled in practice? For determining that question, multiple software developers, who can look back on excessive work concerning software development, were interviewed. They had to answer questions regarding best practices of API design, API qualities and agile software development. The next chapter will refer to the content of the interview itself. The full interview guide can be found in appendix A. As a reference served a guidance from the ETH Zuerich (see (Mieg, 2005)), however I did not adopt all of it. For a recap, this chapter has the following purpose:

1. Validation of some practices identified in chapter 1.
2. Generation of ideas of synergy effects for the combination of API management and agile approaches

3.1. Interview Structure

This interview consists of three parts:

1. **Systems with APIs and their Adaptations:** This part is seen as introduction and the questions deal with general aspects of software systems with APIs and adaptations of APIs.
2. **Determining Best Practices of APIs:** The interviewed persons are faced with best practices for designing a good API. They are able to express their opinion to each best practice.
3. **Review and Outlook to Agile Development:** Includes a recap and ideas about how agile software development can improve the API development.

3. Expert Interview

3.2. Interviewees

I have chosen persons, who possess many years of experience in software development and especially have a deeper knowledge in API development. All interviewed persons are working in software development at AVL. The results of the interview of each person are published anonymous. A list of all interviewed persons can be found in appendix B. *I-1* stands for the first interviewee, *I-2* for the second one, ect. This numbering has no relation to the provided list of interviewees in the appendix for ensuring the anonymity.

3.3. Interview

This section deals with explanations and personal expectations of the interview guide.

3.3.1. First Part - Systems with APIs

Interviewing Process

Before the persons will be faced with the first question, they receive the definition of an API by Martin Reddy (see (Reddy, 2011, p. 4)). This is mandatory, to keep the scope limited for different interpretations. After that, the following questions will be asked:

- **a:** In your opinion from the developer's point of view, what are the advantages of systems, which are using APIs in contrast to systems, which do not?
- **b:** Have you ever participated in the development of an API?
- **c:** Have you ever faced a situation, in which an adaptation of an API would make sense?

Purpose of the Questions

The intention is to provide a smooth introduction to the topics, by thinking of general questions of API management. The second question reveals and confirms whether the person has a deeper knowledge of API development. The last question should figure out whether the interviewee has ever faced a negative experience by using an API.

Personal Expectations

My personal expectation to the first question was, that every interviewed person will list a few advantages, especially the terms *Encapsulation* and *Modularization* should occur. In my opinion, every experienced developer should recognize at least those benefits, because they improve the structure of a code. Concerning the question whether the person has participated in a development of an API: I assumed that a positive answer would be given, because of the experience of each participant.

3.3.2. Second Part - Determining Best Practices

Interviewing Process

The second part consists of two open questions referring to a well designed API, a bad designed API and of five best practices of API design. At first they will be confronted with the following open questions:

- **a:** Please move back to a situation, in which you had to try an API. You got, what you expected, the API worked well. Please describe the properties of this specific API, respectively- in your opinion, what are the qualities of a well designed API?
- **b:** Please move back to a situation, in which you had to try an API. In this case you did not get, what you expected, the API did not work well. Please describe the properties of this specific API, respectively- in your opinion, what are the qualities of a bad designed API?

3. Expert Interview

When those two questions are finished, then a switch to the following five best practice guidelines is performed.

1. **c:** Do not expose member variables, which are not part of the interface
2. **d:** Develop an API as a software system with a low coupling and a high cohesion
3. **e:** An API should be self-explanatory, there is no need for documentation
4. **f:** If a redesign of the architecture would need too much effort, try to implement a wrapper of the interface by using a Proxy, Adapter or a Facade pattern
5. **g:** API testing should be achieved at least with Unit and Integration tests

Every participant was able to make his or her own views known according to each guideline by choosing between the following answers:

- I agree totally
- I agree
- I do not agree
- I totally do not agree

Of course they were allowed to give a full explanation too.

Purpose of the Second Part

The aim of the first open question, which was about a positive experience with the usage of an API, was that the interviewed person shares his view about properties of a well designed API. To obtain very detailed facts, I wanted that the person thinks of a specific situation, which he or she experienced. The second question was about the contrary content of the first question and should also serve as confirmation of the first one. The survey of the five best practices aims at validating the findings of the theoretical aspects identified in chapter 1.

Personal Expectations

I was persuaded that every person has experienced a positive usage and a negative usage with an API. Additionally I was sure, that they would explain the desired properties of a good and a bad designed API in detail, with the experienced situation in mind. With regard to the best practices: I have assumed that every participant will agree to each guideline, because they emerged from research and I wanted to test if these practices are also deemed relevant by practitioners.

3.3.3. Third Part - Review and Outlook

Interviewing Process

The last part of the interview tries to link API development and agile approaches and to recap the five guidelines.

- **a:** Do you see any risks by applying the five best practices?
- **b:** Do you think, that diverse problems could be avoided by applying agile software development? If yes, which ones?

Purpose of the Third Part

The first question tries to elicit doubts about applying the five guidelines. The last one deals with agile software development practices. It is interesting to find out the the opinion of each interviewed person, whether they believe that agile software development could avoid some problems.

Personal Expectations

I have expected that nobody has any doubt for applying any of the five practices. With regard to the agile software development, I assumed that everyone would be critical towards to the agile practices.

3. Expert Interview

3.4. Interview Answers

In the following sections, the answers of each question will be provided.

3.4.1. First Part - Systems with APIs

Questions:

a:

In your opinion from the developer's point of view, what are the advantages of systems, which are using APIs in contrast to systems, which do not?

b:

Have you ever participated in the development of an API?

c:

Have you ever faced a situation, in which an adaptation of an API would make sense?

Answers:

- **a:** The first interviewee is really sure, that there exist no systems without any API, but according to him, he knows examples in which APIs are badly designed or misused. He underpinned the importance of APIs with the metaphor of a car: without the concept of an interface, it would be impossible or at least really difficult to drive it, because then there exists for instance no key to start it- so the car has to be started in another way. In detail he noted that the maintenance and the usage of a software component will be improved through the use of an API, because it decreases the coupling, hides the implementation and an user of an API does not have to think about this implementation (I-1). The second interview partner is persuaded that every system uses some kind of an API. He thinks that the weak rigidity of an API (the ability to change it), must not be wrong in principle, because it enables the API to adapt according to the conditions. Using an API gives more stability and it makes it possible that different teams can work simultaneous, because of dividing and assigning the tasks to

multiple teams (I-2).

The third one is convinced that an API is mandatory if there is a loose customer-supplier relationship (e.g.: if there exists no joint development), so durable agreements would improve the development process. He determined a huge possible disadvantage by using APIs internally for the simultaneous development between different teams: if one team has to perform a work-around and therefore to adapt an API, then they have to prepare a new agreement with the other teams. This could be a dampening effect (I-3).

The fourth interviewee mentioned among others the contract between the client and the company, which has to fulfil the contract and that an API hides the implementation. Additional advantages would be a separate deployment in diverse systems, the thoughts of versioning and maintaining backward compatibility because multiple clients may depend on that API. An additional aspect was testing, because defined interfaces are well testable and it can be determined whether a problem exists within or outside of the interface. This implies an increasing quality of the software (I-4). The last interview partner points to the fact that the existence of an interface contract, between two parties represents a huge advantage. This enables teams to adapt their work according to the expectations, which are defined in the contract (I-5).

- **b:** All five interview partners were involved several times in the development of APIs (I-1), (I-2), (I-3), (I-4), (I-5).
- **c:** Again, all five interview partners faced such a situation. (I-1), (I-2), (I-3), (I-4), (I-5).

3.4.2. Second Part - Determining Best Practices

Questions:

a:

Please move back to a situation, in which you had to try an API. You got, what you have expected, the API worked well. Please describe the properties of this specific API, respectively- in your opinion, what are the qualities of a well designed API?

b:

3. Expert Interview

Please move back to a situation, in which you had to try an API. In this case you did not get, what you have expected, the API worked not well. Please describe the properties of this specific API, respectively- in your opinion, what are the qualities of a bad designed API?

c:

Do not expose member variables, which are not part of the interface

d:

Develop an API as a software system with a low coupling and a high cohesion

e:

An API should be self-explanatory, no documentation should be necessary

f:

If a redesign of the architecture would need too much effort, try to implement a wrapping of the interface by using a Proxy, Adapter or Facade pattern

g:

API testing should be achieved with Unit and Integration tests

Answers:

- **a:** The first interviewee mentioned that there are good examples found in the ".NET" framework. Important properties of a well designed API are *intuitive (self-explained)*, *documented* and *consistent*. In this example, the chosen development approach was an agile one, which leads to a positive impact on the API (I-1).

The second one stated, that it is unlikely that there will not occur any problems by using an API in practice. He believes that *less complexity* leads to more stability and fewer problems, in this context he refers to GUI interfaces, which are examples of complex APIs. Additionally, a *separation of responsibilities* and a *reliability* of the API (no bugs, an API should always do what it is supposed to do) should be performed. He argued that the API was developed with agile methods and mentioned that *iterative approaches*, *refactoring* and *test driven development* lead to a good API (because tests are the first clients of the API). It was referred to test driven development again, because by writing tests first, it is possible for the developers to put themselves in the place of an user of the API. After that, it was noted that the API should be intuitive.

3.4. Interview Answers

He stated also, that the first version of an API will probably not be the best one but iterative approaches with frequent refactoring and test driven development will have a positive impact on an API (I-2).

The third interview partner mentioned the ".NET" framework as a good example for a well designed API too. In detail, desired properties are: a *clear structure of the namespaces*, *self-explanatory*: there is no need to read the corresponding documentation, *modularization* and *self-explanatory names*. The development of APIs, in which he participated, was not performed with any specific development approach (I-3).

The fourth stated that a well designed API should be *extendable*, *standardized*, *documented* and adaptations are only made during the *versioning*. The development of an API, was not agile from a formal point of view, but it included some agile practices (I-4). The last one claims that an API has to be *well structured* and *documented*. Furthermore, a well thought-out error handling has to be provided too, for that the specific methods have to be identified by their corresponding errors. The development process was also not a declared agile one, but there existed small beginnings of agile practices (I-5).

- **b:** There was a clear statement of the first interviewee, that a well defined API solves a problem immediately and a bad defined API will not solve the problem or the developer has to spend much more time for figuring out what to do. A developer will face a problem, if the API is *not intuitive (self-explained)* and if there exit a *lack of documentation* (I-1).

The second one stated that problems might occur if an API is *not stable (needless problems, caused by adaptations of an API)*, *not self-explained*, *not modularized (too high degree of coupling caused through an unclear division of responsibilities)*, *not well documented* or if it *performs not for what it was made for* (I-2).

The third interview partner mentioned *too few thoughts about useful namespaces*, *no logical division of functionalities with regard to the namespaces (bad modularization)*. It is not a good practice too, if *too many classes or functions are set to public unnecessarily* (I-3).

The fourth interviewee stated that an API should not be *too complex* regarding the usage, caused through varied options and there should not be a *lack of documentation* (I-4).

3. Expert Interview

The last one thinks that a badly designed API *does not provide any documentation and changes by releasing a new version* e.g.: a different calling convention. It is also a bad practice, if there exists *no provided test* for the API and *no explanation* about the adaptations. The worst thing which can be done is to *break the contract*, e.g.: caused by adaptations, this will have a negative impact to all who are using this API. Another cautionary tale is not to consider the need of the customer: e.g.: an API will not be developed in the interest of a customer (I-5).

1. **c**: I agree totally (I-1) (I-2) (I-3) (I-4) (I-5)
2. **d**: I agree totally (I-1) (I-2) (I-4) (I-5), I agree (I-3)
3. **e**: I agree (I-1) (I-2) (I-3) (I-4) (I-5) *
4. **f**: I agree totally (I-5), I agree (I-1) (I-2) (I-3) (I-4)
5. **g**: I agree totally (I-1) (I-2) (I-3) (I-4) (I-5)

* A note to the third guideline has to be done, because every interviewee thinks that this is theoretically a good practice, but it is probably not possible to achieve this for every API. E.g.: it is not possible to keep all methods self explanatory, this could lead to rather long method names. Therefore, not everything can be expressed intuitively.

3.4.3. Third Part - Review and Outlook

Questions:

a:

Do you see any risks by applying the five best practices?

b:

Do you think, that diverse problems could be avoided by applying agile software development? If yes, which one?

Answers:

- **a**: The first interviewee believes that in general, there are no risks (I-1). The second one mentioned problems concerning the software quality of parts which do not belong to the API, by strengthening the stability of the API- especially an *aversion to adapt an API* even if the whole

would be improved (I-2).

The third interviewee states that there are no risks for applying those practices in general, except for the fourth and the last guideline. Regarding the fourth guideline about wrapper interfaces he believes that it is situation related, whether it would be a better solution to realize a redesign, instead of wrapping the API. There are also concerns about automatic tests, if the focus of the implementation of a huge amount of automatically functional tests, may lead to the phenomenon that manual operations would be neglected (in the context that nobody has a knowledge about how the corresponding GUI works) (I-3).

The fourth interview partner believes, that there will not occur any problems by applying the five practices (I-4). The last interviewee thinks that developers, who do not have a test driven background, will neglect testing (I-5).

- **b:** The first interview partner stated, that there exists just one big benefit of agile methods: this would be to "fail fast". He is not persuaded, that agile methods lead to a faster development and to more efficiency, but it should be recognized early enough through *quick feedback*, whether the project is on the right track (I-1).

The second one stated that agile practices prevent a team to define an API "upfront": It avoids among others, that the requirements of an API will be finalized at the beginning and will not be changed during the development, by getting *quick feedback* and by *using test driven development*. There should be no fear to change an API before the release, if it will improve the whole system (I-2).

The third interview partner stated that Scrum could help to *keep the balance between manually and fully automated tests*. Applying agile practices may lead to a more *frequent refactoring* of the software. There is definitively a benefit for APIs, because clients and developers have to *communicate more frequently*. With the classical approach, it is probably impossible to reach a working complete system, by agreeing on an interface and no communication during the implementation for a long time. It should be mandatory to make a *frequent review* concerning the desired functionality (I-3).

The fourth interviewee mentioned *immediate feedback, simultaneous development (fewer dependencies during the development by agreeing on the contract)*, as huge advantages (I-4). The last one thinks that a more

3. Expert Interview

stable contract between two parties will be achieved through shorter delivery cycles and early feedback (I-5).

3.5. Conclusion of the Statements of the Interviewees

In this chapter, the statements of each interview partner will be analyzed.

3.5.1. First Interviewee

The assessment of the first interview partner is, that the implementation of APIs in systems should be mandatory. The most important properties are among others, that APIs are self-explaining, documented and consistent. There exists a very critical approach about the first usage of APIs, because if there occurs an uncertainty, an API will be rejected immediately. Additionally, if the result is not similar to the expectations, an API will be declined too. The opinion to agile development approaches is as follows: at first it was noted, that performing an agile approach would have a positive impact on an API, but there exists no total conviction. All five practices should help to realize a well designed API. Agile approaches may help to improve the positive impact of an API, but they have to be treated with care. Intensive testing is realized by test driven development. A self-explanatory API can be reached with an intensive cooperation with the client and among others refactoring and test driven development will support to reach a consistent system.

3.5.2. Second Interviewee

It seems that this interview partner has a deep knowledge about agile practices, software development and their close cooperation. This is indicated through his frequent references to agile practices and how they will influence the API development. E.g.: Test driven development should be applied

3.5. Conclusion of the Statements of the Interviewees

because this practice works as first tests of the clients. Frequent refactoring will improve the API, because at the beginning no API is perfect and early feedback through an iterative approach is a positive impact too. The focus on practical solutions is remarkable too, therefore that it was noted that reliability, less complexity and a clear separation of responsibilities would be suitable properties of an API. There is no doubt that a system can exist without using any API. A general consent exists to use all practices for improving an API. Many statements of this interviewee implied that an API will evolve and improve through time (adaptation according to the conditions, iterative approaches with frequent refactoring and test driven development will have a positive impact on an API, agile practices prevent a team from defining an API "upfront", aversion to adapt an API even if the whole would be improved) and that agile approaches will support that.

3.5.3. Third Interviewee

Benefits of using APIs are seen concerning the relationship to the customer, if there exists no joint development by meeting the contract. A negative impact could occur through an evolving damping effect caused by an internal adaptation of the API- so teams which depend on that API have to wait until the changes are done. Explicit properties of a well designed API are self-explanatory, modularization and a clear structure of the namespaces. Negative properties would be badly chosen namespaces, no logical division of functionalities with regard to the namespaces and if classes or functions are unnecessarily set to public. Like the first and the second interviewee, this interview partner agrees with all five practices. Agile practices could have a positive impact on API development, because of a more frequent refactoring, an improved communication between customer and developer and a better balance between manually and fully automated tests.

3.5.4. Fourth Interviewee

The defined contract between the customer and the company, implementation hiding, a possible separate deployment in diverse systems, versioning

3. Expert Interview

and the maintenance of backward compatibility, a better quality of the software and an improved testability are seen as advantages. This interviewee demands that a well designed API has to be extendable, standardized, documented and that adaptations are only made by versioning the API. Contrary, no API should be too complex with regard to the usage and there should not be a lack of documentation. All five practices seem to be okay. Agile approaches should lead among other things to an early feedback and support simultaneous development.

3.5.5. Fifth Interviewee

The last interview partner mentioned the contract as a positive impact, so that both parties could adapt their work according to this contract. Definitive properties of a well designed API are a good structure, an available documentation and an intelligent error handling. On the other hand, bad examples would be a lack of documentation, no explanation of adaptations, which may cause a break of the contract between the customer and the company. All five practices should theoretically lead to an improved quality of an API. Agile software development will have a positive impact on the stability of the interface contract, to achieve an early feedback caused by shorter delivery cycles.

3.6. Conclusion

This conducted expert interview confirmed selected aspects of chapter 1. Basically, all interviewees have rather practical views about API development and agile approaches. Among others, the purpose of an API should be clear, it also has to be stable, easy to apply and documented. All five interview partners are convinced about the fact that agile approaches would have a positive impact to the achieving quality of an API. An aspect which should be highlighted in my opinion is the support of the communication between customer and developers.

4. API Development and Agile Software Development

This chapter deals with the question how agile approaches, which were discussed in chapter 2, could improve the development of high quality APIs-see chapter 1. Some practical understandings are based partially on chapter 3, in which I have conducted an expert interview with experienced software developers, combined with my personal deductions. Furthermore, some synergy effects between API development and agile approaches are based on a literature research. This chapter links many aspects of the previous chapters, for finding synergy effects between API management and agile software development.

4.1. Positive Impact by Applying Agile Approaches

I will analyze each mentioned positive impact of agile approaches, listed in chapter 3 and I am going to derive personal annotations why they can support API development. The listed benefits are based on the literature research of chapter 1, chapter 2 and the expert interviews in chapter 3.

4.1.1. Quick Feedback

Quick feedback, which is also discussed in chapter 2.1.2, is essential for rapid reactions concerning to steering the project in the right direction. How can quick feedback be ensured? From the view of the customer, which is

4. API Development and Agile Software Development

the most important one, there is a need for a working software- a customer will probably not possess as many technical skills as the developer. So there has to be a running software at the end of each iteration- a way for supporting that is by applying continuous integration and continuous delivery. Continuous integration ensures that the software will improve in quality and get additional functionality by every merge and the concept of continuous delivery promises to deploy a working software rather fast, like clicking on a button. A quick feedback could probably avoid the situation in which an API is released but at the bottom line it does not fit according to the requirements.

4.1.2. Cooperation of Customer and Developer

Agile approaches require that developer and customer have a close cooperation and meet daily, as mentioned in chapter 2.2: this is necessary that customers can share their vision and their thoughts on the software with the developer, so they do not have to wait for a possible meeting, which will be held in a few weeks. They can react immediately, telling their concerns and suggestions about an API. Therefore, developers will not lose much time, they would have spent working in a wrong direction. As already mentioned in chapter 2, this aspect of a close cooperation is an agile principle and this is a good thing. The API will be used by the customer, they want to solve their individual problems by applying the API and satisfying a customer is an agile principle too. As mentioned before, this cooperation belongs to one of the agile principles. The benefits are obvious: in conclusion, the customer has to be satisfied, that is the most important aim. For this reason, the API has to be adapted according to the needs of the customer. Requirements can change rather fast, therefore frequent communication is essential. It is about an API and not about a simple application: once an API is released to the customer, adaptations should only be performed "undercover", customers must not be contacted, as discussed in chapter 1.2.3. They should just notice, that an API has been updated but they should be able to use it as they have done it in the past.

4.1.3. Stable Contract

An interface contract, mentioned among others in chapter 1.1.2, defines what developers are supposed to deliver to the customer. Both parties will prepare and maintain this specific contract. Through agile approaches, this is not a kind of contract, which will be created one time and after that, developers will work on the API until it is finished, according to this contract with no intermediate communication. This approach would probably not lead to a satisfied customer, because requirements will change and for this reason the contract has to be updated too. Agile approaches (see 2) have a positive impact on that, because they offer frequent development cycles and the different versions of the product will be presented to the customer, so an early intervention is possible.

4.1.4. Evolving API

After releasing an API, changes should only be achieved "undercover" (see 1.2.3). This means that the implementation can be changed, but not the interface. Performing adaptations to the interface, e.g.: the calling convention of a method would result in interface breaks and in the next days, the customer will probably call the company, complaining about problems with their product (e.g.: no successful building after updating to the new API) or complaining about workarounds they had to achieve because of the changed API. Those are worst case scenarios: a customer should never be aggravated. So, the point in time, when the API will be released for the first time, is a very critical point- as described in chapter 1.4.5. After that, adaptations have to be done very carefully. The basic framework, that future adaptations can be done easily after the release of an API, is to rework the API as much as possible, if there exist concerns about it. The first version of an API, before the date of the release, will not be perfect. Of course not, but no developer should be afraid to make fundamental changes in the time before the release, if they are assumed to be right. If huge adjustments have to be done, then they should be done before the release. Agile approaches assume that refactoring work is done during the different development

4. API Development and Agile Software Development

cycles. Additionally, feedback of customer and colleagues should have a positive impact on promoting an evolving API too.

4.1.5. Refactoring

Refactoring is an essential work, without it, code tends to become more and more fragile, code will become cryptic for colleagues and nobody feels confident to adapt code, which is not understandable. Customers expect a stable and working API (see chapter 2.2), for this reason code which is not usable for maintenance is a terrible image. Agile approaches prevent that, by promoting refactoring. Companies, which do not integrate agile approaches will maybe not take advantage from that, because if many programmers will do just their own work without asking for assistance, then their software will probably not reach a higher level of quality.

4.2. Properties of a Well Designed API and their Relation to Agile Approaches

In this section I will list and analyze properties of a well designed API, which I have experienced in chapter 3.4, and determine how agile approaches (see chapter 2) can help to perform these properties. I will analyze why every mentioned property of a well designed API will improve an API and in what way agile methods, practices or principles could influence it. The following explanations are also my personal conclusions.

4.2.1. Simple

An API should be simple, as discussed in chapter 1.2.5, so as little complexity as possible is desired. Customers prefer a more simple API in contrast to a complex one, because every customer just wants to finish his job as easy as possible. In general everybody wants to solve a problem rather quickly, therefore complicated problem solving approaches will be avoided. But

4.2. Properties of a Well Designed API and their Relation to Agile Approaches

how can "simplicity" be defined? It is in the eye of the beholder, in that case the most important reference person is the customer. In the best case, there is a frequent communication with the customer, a close cooperation to keep steering in the right direction, which is part of the agile principles, mentioned in chapter 2.2. According to the active development, it would be a good idea if developers think as customers. Simplicity, cooperation of customers with developers and satisfying customers with a working software are agile principles. For that, agile software development demands that developers have frequent communication with their customers, and they will give feedback to the developer. Iterative development cycles will support simplicity too, because everybody is allowed to give a feedback. A test driven development approach will also have a positive impact. This approach requires to write tests before the corresponding code is written. By doing that every developer is supported to step into the shoes of a customer. For that, the easiest way for applying an API will be considered.

4.2.2. Documented

A documentation, discussed in chapter 1.2.7 should at least include explanations to aspects of an API, which could not be expressed through e.g.: an intuitive naming. In the best case, every small detail is explained. If an API has to be maintained, every developer who has the honor to perform some changes will be glad to have a detailed documentation. The development should be sustainable, this is required by an agile principle (see chapter 2.2). Therefore it is necessary that every developer has as much knowledge about an API as possible. Nobody will gain a deep knowledge of every part of a system, which consists of multiple classes and many different methods. For this reason, documentation is necessary.

4.2.3. Consistency

This describes to agree on a concept (see chapter 1.2.5), which has to be present in every part of the system. E.g.: the project team agrees that each method should have no more than three parameters. The idea of

4. API Development and Agile Software Development

this concept has to be present in every module, every class and in every method. This improves every system, because every developer should agree on this idea and perform his individual work according this concept. A colleague will be able to understand another part of the system more easily too, by considering the concept. In contrast to a classical approach, the agile one demands a frequent refactoring, so the API will often be improved. Another aspect are the reviews: everybody is allowed to make suggestions about improvements- see chapter 2.2. Daily meetings should also support consistency, because everybody will get notice about the work of the colleagues. It will be more likely to determine a deviation by a more frequent communication.

4.2.4. Self Explained

It is really annoying, if a problem can be supposedly solved with a provided API, but by applying the API, a customer has no idea which e.g.: method has to be applied or if the method requires a high amount of arguments with cryptic names. The customer will probably loose interest in working with this API, using the documentation for figuring out what to do is not a popular pastime. For a recap, the attribute of self explanation is discussed in chapter 1.2.5. Agile approaches could have a positive impact on this aspect too, because frequent communication with the customer, early feedback and reviews could improve an API to become more intuitive.

4.2.5. Reliability

An API has to be reliable which is also mentioned in chapter 3. Nobody will use an API for a longer period, if an API does not work as suggested or if workarounds have to be performed for fixing the whole system because of the unstable API. A non reliable API will be deprecated, as discussed in chapter 1.4.5. For avoiding that, an API has to fulfill its purpose, and has to be stable. The key usage can be identified by a close cooperation with the customer, which agile approaches require. Another agile approach demands an intensive testing: by applying test driven development, the API will be

4.2. Properties of a Well Designed API and their Relation to Agile Approaches

proven intensively because of the high degree of code coverage. This will be reached because of the requirement that tests have to be created before the corresponding code is written.

4.2.6. Integration of Adaptations only through Versioning

In my opinion, one of the characteristics of a good API is that it can be used for a long time. Nobody wants to change an API rather frequent, this implies much work. Of course a software is never finished, therefore an API has to be maintained too and probably new features will arise as discussed in chapter 1.4. Therefore it is important that adaptations are clearly declared. A versioning process, mentioned in chapter 1.4.5 could support this aspect. The customer knows, that a new version number will imply a new version of an API, which is in some way changed. The kind of adaptation may also depend on which kind of version number was changed, e.g.: by using a "Major-Minor-Patch" versioning system. Ideas of adaptations evolve through communication with customers and with colleagues. Customers will probably always find new suggestions to apply an API easier for them. Agile approaches, see chapter 2, require to provide a release plan. Within the scope of such a release plan, adaptations or new features will be proposed. For that, a change to the API will be done in a structured way.

4.2.7. Extendable

Software is never finished, for this reason an API will also evolve over time. It is very important that it can be extended with new functionality and keep stable. Large software projects tend to become fragile as discussed in chapter 1.2.4, if adaptations and new features are not developed in a way to keep the whole system stable. The introduction of APIs force encapsulation and modularization. A problem could occur if e.g.: a method cannot be extended, because if this will be done, then backward compatibility may be broken, as mentioned in sub chapter 1.4. Therefore, every part of an API has to be developed with care accordance with future development.

4. API Development and Agile Software Development

Frequent refactoring and reviews, which are required by agile approaches, see chapter 2.3.2, could support those aspects. A very experienced developer may think of possible extensions of a method and keep that in mind.

4.3. Interactions of Agile Approaches and API management Based on Literature Research

4.3.1. Collaboration of Developers with QA Engineers in a Scrum Sprint

In the best case, developers are supported by a QA team, which is among others responsible for writing integration tests. So, the accountability of writing automated tests is shared between the developers and the QA engineers. It is important to notice, that the interaction with a QA team depends highly on the software development model. There is an immense difference between a traditional and an agile approach: According to the traditional development process (e.g.: waterfall model), testing of software is the last step. It is obvious that there is often not much time left, because of diverse problems in the previous steps, but the QA team has to do their work: proving the quality of the software. In this model, doing the quality assurance is an ungrateful job, because apart from the aspect of time pressure before the release, the justified work of this team will delay the finish of the work. This inflexible approach of the waterfall model can lead to further delays and to a negative image of the QA team.

A more agile way can help by including the QA engineers into the development process and adding testing activities in each iteration or sprint. This will lead to an earlier recognition of quality problems and the management and the developers are able to take countermeasures. One possible agile development process would be Scrum (Reddy, 2011, p. 304-305).

4.3. Interactions of Agile Approaches and API management Based on Literature Research

4.3.2. Determining Key Usage of an API

As mentioned in chapter 1: an API is a contract between the company and the clients. The creator of an API provides an interface to the customer for using some functionality. If the customers are satisfied with the API regarding different factors as usability, functionality- then the target was reached. In some cases, it would be an advantage to undertake the role of the customer, for designing a high quality API from another viewpoint (Reddy, 2011, p. 305).

Test Driven Development Approach

For this purpose, test driven development can be applied, because writing tests before the corresponding code is implemented, may lead to a focus on the key usages of the API and there are only features implemented which are really needed. In that way, applying test driven development will address the concepts of chapters 1.2.4 and 1.2.5 (Reddy, 2011, p. 305).

Scrum Approach

Another agile method might help: Scrum (see Ken Schwaber (2013)): Instead of just providing a list of detailed requirements, which will be implemented, Scrum demands to determine functionalities from the view of a user. Those functionalities will be implemented iteratively during the specific sprints (Ken Schwaber, 2013). As already mentioned, designing an API from the viewpoint of a customer would be a huge advantage (Reddy, 2011, p. 305).

Extreme Programming Approach

As described in the previous chapter, applying Test Driven Development leads to a better way of determining the key usage of an API. The real domain of an API could be gathered earlier by applying Extreme Programming.

This statement is based on the fact, that Extreme Programming requires to

4. API Development and Agile Software Development

implement the role of a customer. This customer has a permanent interaction with the whole team and provides the requirements of the product. An important aspect is, that the customer is present through the whole development process until the delivery of the product, instead of just defining the requirements at the beginning of the development. He can steer the qualities of the product continuously, because the XP team fabricates the product in several small releases. That is based on a rudimentary release planning at the beginning and several iteration plannings throughout the whole development process (Jeffries, n.d.). If the deadline of the release is soon and not all features are implemented, the customer can determine, which of the standing functionalities should be realized, by prioritizing them. By applying XP, test driven development is also performed, which supports the development process and this leads to an API of a higher quality.

4.3.3. Realizing Aspects of Coupling by Applying Design Improvement of XP

As mentioned in chapter 1.2.6, it is a good advice to design an API with a high degree of cohesion and a low degree of coupling. This is based on the fact, that the reverse will not allow single components to be independent from other components. An adaptation of a component implies an adaptation of a dependent component. With this approach, encapsulation should be improved, making it possible to reuse single components (Reddy, 2011, p. 52-53). The practice "Design Improvement" of the XP approach realizes this advice. The goal of Design Improvement is to provide a well designed software, for this reason refactoring is performed. Using Refactoring consequently leads to a software which does not contain duplicate code and has the desired property of a low coupling and a high cohesion (Jeffries, n.d.).

4.3.4. Face to Face Conversation Applied in API Reviews

Face to face communication is one of the most important agile principles, because it facilitates a dynamic way of information exchange amongst

4.4. Conclusion

developers. If conversations were held frequent and in a high quality, then the requirements will be emphasized. That is the agile way (Apke, 2014).

For this reason, the agile principle of the "face to face conversation" should be applied in the whole API development process, especially at reviews, pre-release reviews and communication with customers. A misconception in any stage of API development may have fatal impacts, because of the fact that an API has a high importance for the success of a company.

4.4. Conclusion

Theoretical aspects promise that agile software development approaches will have a positive impact on software development. This chapter provided some examples, why agility could improve processes and therefore software would achieve a higher quality. API development is a kind of software development, so why should there be a difference because of applying agile software development approaches? API development includes a very critical part, which could be a deciding factor for the success of the company: this is the date of the release of an API. After this point in time, the created concept of the API has to be pulled through, until the API is removed after publishing a deprecation. The deprecation of an API, with their resulting removal should not be performed too soon, an API should be developed for a long-ranging use, because the development includes much effort. There should be a high focus on the time before the release. A close cooperation with the customer should be performed, thinking about simplicity, coupling aspects, ect. To me, a successfull development of an API is reached if the customer is satisfied and if the maintenance and extension of this API is not too much effort to the project team.

5. Tools for Determining API Breaks

With my diploma thesis, I have developed "ApiDiff" and "AssemblyParser", two C# applications- which should support a continuous prove of compatibility between different versions of specific products. The aim of those tools, which are applied frequently at AVL, is to find out which API changes were made during a new release of an AVL product. For that reason, build breaks, caused by violations of backward compatibility, should no surprise any more. The development of those tools triggered the question about how APIs should be designed for preventing such a situation and furthermore, how agile software development could support this.

5.1. ApiDiff

5.1.1. Term Definition

- **Platform:** A software product of AVL, which serves to provide functionality to other AVL products.
- **Artifactory:** Binary repository manager for providing interface packages, see <http://www.jfrog.com/video/artifactory-1-min-setup/>
- **QA:** Abbreviation for "Quality Assured" only tested software products will be stored at this specific part of Artifactory
- **Snapshot:** Products in development will be stored at this part of Artifactory
- **Product 1:** A software product, which uses APIs of the Platform
- **Product 2:** A software product, which uses APIs of the Platform

5. Tools for Determining API Breaks

- **Interface Packages:** Compressed collection of different library files (.dll). Common file name of those packages are the name of the product, followed by a major number and a version number. A Platform specific interface package would be named e.g.: "Platform-1.2-876.0-Interfaces.zip"

5.1.2. Motivation

Current Situation of the "Platform"

The Platform implements functionality for several products, which are named Product 1 and Product 2 in this context, for avoiding developing redundant code. Those products depend on the Platform, because of using some functionality. By releasing a new version of the Platform, violations according to backward compatibility may occur- in many cases because of changed APIs, which causes interface breaks. Adaptations of the Platform may have direct influences to those products. A compilation of e.g.: Product 1 against a new version of the Platform often does not succeed, e.g.: because of an changed API of the Platform, which is used by Product 1. An idea of a tool arised, which is able to detect potential interface breaks. It should be possible for detecting at which specific location the break was noticed. If a developer has done a change, he can prove by using ApiDiff, whether his release maintains backward compatibility.

A more precise description of the problem can be illustrated by considering figure 5.1: components of the Platform are available in several different releases, stored in binary at a repository manager, called Artifactory. Quality assured releases are stored at QA and test versions are stored at Snapshot. By releasing a new version of the Platform, it has to be assured, that backward compatibility is maintained. Violating API compatibilities leads to a build break of the dependent products, because they will not compile against the new version any more. For that, ApiDiff should determine API compatibility relevant changes of a new version of the platform according to a quality assured version (consider red marking in figure 5.1).

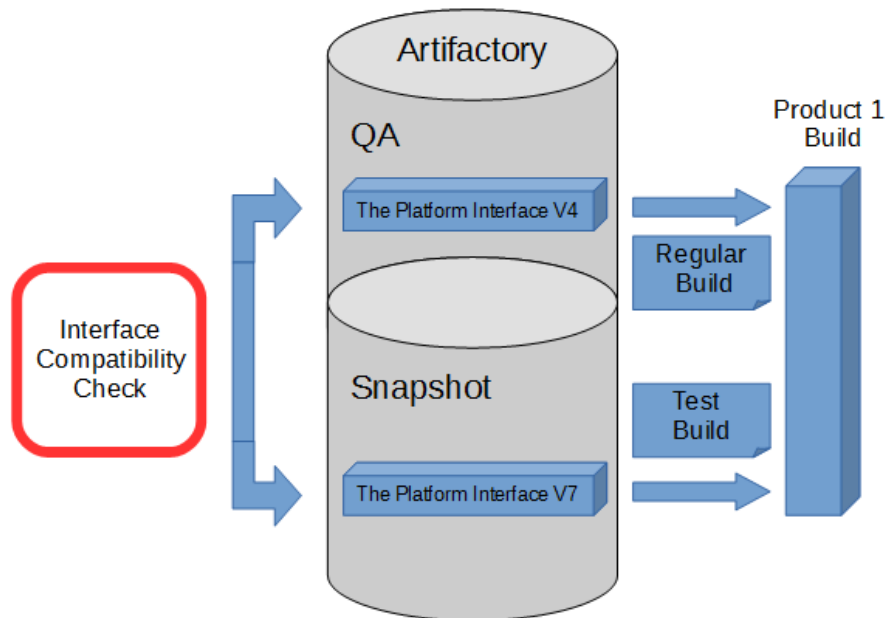


Figure 5.1.: Current situation of the platform according to build break

Purpose of this Tool

The objective is to detect build breaks by continuous checking of interface compatibility of the Platform. It should be possible to determine differences between different releases of the Platform. This is solved in a binary way by processing changes between library files, e.g.: which classes are added or removed, which methods have been changed according to one pair of assemblies. The tool provides an overview about the critical changes of interfaces (e.g.: removed classes, removed methods) and about which classes, interfaces and their corresponding methods have a relatedness to Product 1 or Product 2. Assemblies, which will be provided by Artifactory will be scanned and so called report files (XML files) will be created.

5. Tools for Determining API Breaks

5.1.3. ApiDiff in Collaboration with Artifactory

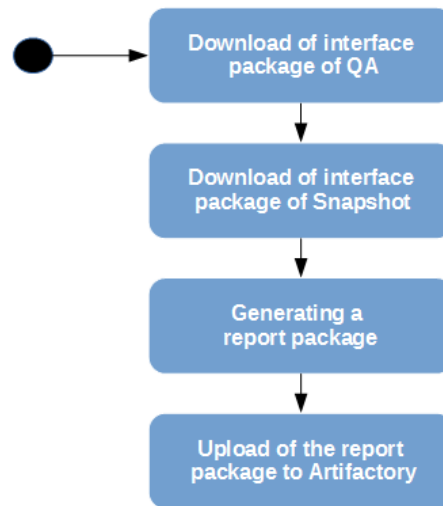


Figure 5.2.: Process Sequence of ApiDiff in Artifactory Mode

For the usual disposition (Artifactory Mode) of ApiDiff at AVL, a collaboration of the tool with Artifactory, as seen in Figure 5.3, is necessary. In this specific mode, ApiDiff fetches different interface packages (depending on the passed parameters) of Artifactory (e.g.: Platform-1.2-876.0-Interfaces.zip”). Interface packages of the QA repository are reference versions, interface packages of the Snapshot repository serve as a test versions. After that, processes for extracting, parsing, generating reports and compressing those reports will be started. The final step is the upload of the created report files to the Artifactory server (in this case: Platform-1.2-876.0-Reports.zip”). Such a process sequence can be seen in Figure 5.2.

5.1.4. Types of Reports

In general, there are two types of reports, a “Single” and an “Overview” report. Single reports consists of changes according to a library file pair

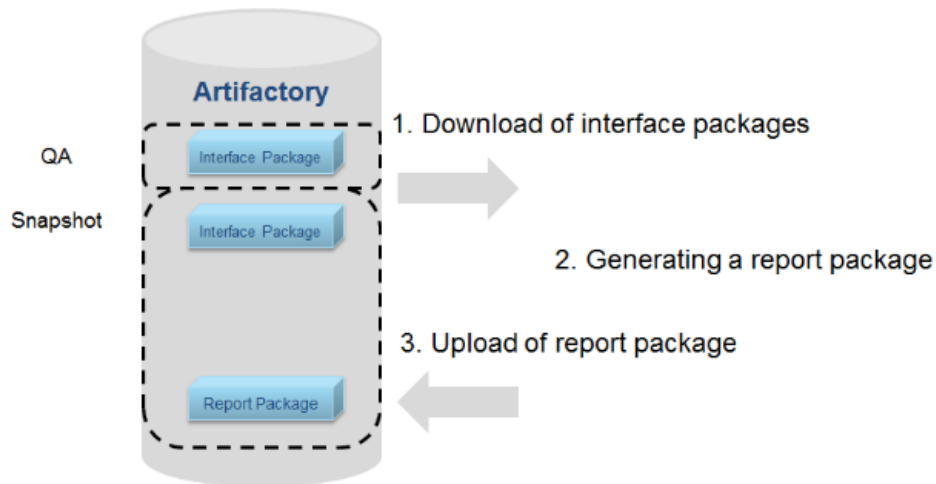


Figure 5.3.: Artifactory Mode of ApiDiff

(old version of a .dll file vs. a newer version of a .dll file). Overview reports notifies about changes according to a whole interface package comparison (an old version of an interface package vs. a newer version of an interface package).

Single Reports

General Description: Shows differences between two single library files (.dll). An example can be seen in Figure 5.4.

Single Reports with Regard to Corresponding Library File Pairs

- **General Reports:** Report_<Name of the Library File>.xml
- **Critical Reports:** ReportCritical_<Name of the Library File>.xml
- **Product Reports:** Report<Name of the Product>_<Name of the Library File>.xml

Description of the XML Tags:

FileInfo: Contains multiple tags and each of them contains general information.

5. Tools for Determining API Breaks

```
<?xml version="1.0" encoding="UTF-8"?>
- <XMLAssemblyPairReport xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  - <FileInfo>
    <FileName>[REDACTED]</FileName>
    <DateOfBuilding>2014-09-08T13:53:40.6657242+02:00</DateOfBuilding>
    <VersionOld>1.2-860.0</VersionOld>
    <VersionNew>1.2-876.0</VersionNew>
  </FileInfo>
  - <DiffSummary>
    <NumberOfChanges>10</NumberOfChanges>
    <NumberOfModifiedClasses>4</NumberOfModifiedClasses>
    <NumberOfRemovedClasses>0</NumberOfRemovedClasses>
    <NumberOfAddedMethods>7</NumberOfAddedMethods>
    <NumberOfRemovedMethods>3</NumberOfRemovedMethods>
  </DiffSummary>
  - <DiffDetails>
    <AddedClasses/>
    <RemovedClasses/>
    - <ModifiedClasses>
      - <ModifiedClass>
        <ClassName>[REDACTED]</ClassName>
        <AddedMethods/>
        - <RemovedMethods/>
          <string>[REDACTED]</string>
        </RemovedMethods/>
      </ModifiedClass>
      - <ModifiedClass>
        <ClassName>[REDACTED]</ClassName>
        <AddedMethods/>
          <string>[REDACTED]</string>
        <RemovedMethods/>
      </ModifiedClass>
      - <ModifiedClass>
        <ClassName>[REDACTED]</ClassName>
        <AddedMethods/>
          <string>[REDACTED]</string>
          <string>[REDACTED]</string>
        </AddedMethods/>
        - <RemovedMethods/>
          <string>[REDACTED]</string>
          <string>[REDACTED]</string>
        </RemovedMethods/>
      </ModifiedClass>
    </ModifiedClasses>
  </DiffDetails>
</XMLAssemblyPairReport>
```

Figure 5.4.: Example of a Single Report

- **FileName:** Name of the library file
- **DateOfBuilding:** Date of the creation
- **VersionOld:** Old version of the library file
- **VersionNew:** New version of the library file

DiffSummary: Contains a compact overview according to the changes.

- **NumberOfChanges:** This number represents how many total changes were made
- **NumberOfModifiedClasses:** This number represents in how many classes were methods added or removed
- **NumberOfRemovedClasses:** This number represents how many classes were removed
- **NumberOfAddedClasses:** This number represents how many classes were added

- **NumberOfRemovedMethods:** This number represents how many methods were removed
- **NumberOfAddedMethods:** This number represents how many methods were added

DiffDetails: Contains more detailed information of the tag *DiffSummary*.

Overview Reports

General Description

Provide an overview according to a bundle of assemblies.



Figure 5.5.: Comparison of Interface Packages

Overview Reports with Regard to a Bundle of Library Files

An example of an Overview Report can be seen at [5.6](#).

- **General Reports:** OverviewReport_<Name of the Library File>.xml
- **Critical Reports:** OverviewReportCritical_<Name of the Library File>.xml
- **Product Reports:** OverviewReport<Name of the Product>_<Name of the Library File>.xml

5.1.5. Usage Possibilities

ApiDiff can distinguish between three different usage modes:

1. **Single Mode:** compares two corresponding library (.dll) files.

5. Tools for Determining API Breaks

```
<?xml version="1.0" encoding="UTF-8"?>
- <SummaryReport TotalChanges="1195" MajorNumberNew="1.2" MajorNumberOld="1.2" VersionNew="876.0" VersionOld="860.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Statistics MethodsCurrent="15522" MethodsOld="15396" ClassesCurrent="2223" ClassesOld="2200" AssembliesCurrent="42" AssembliesOld="42"/>
  - <ModifiedFiles>
    - <ModifiedFile ModifiedClasses="1" FileName="..." Changes="2">
      <AddedMethods>2</AddedMethods>
    </ModifiedFile>
    - <ModifiedFile ModifiedClasses="1" FileName="..." Changes="4">
      <RemovedMethods>4</RemovedMethods>
    </ModifiedFile>
    - <ModifiedFile ModifiedClasses="3" FileName="..." Changes="54">
      <AddedClasses>8</AddedClasses>
      <RemovedClasses>4</RemovedClasses>
      <AddedMethods>30</AddedMethods>
      <RemovedMethods>12</RemovedMethods>
    </ModifiedFile>
    - <ModifiedFile ModifiedClasses="1" FileName="..." Changes="8">
      <AddedClasses>1</AddedClasses>
      <AddedMethods>7</AddedMethods>
    </ModifiedFile>
    - <ModifiedFile ModifiedClasses="2" FileName="..." Changes="739">
      <AddedClasses>2</AddedClasses>
      <RemovedClasses>1</RemovedClasses>
      <AddedMethods>372</AddedMethods>
      <RemovedMethods>364</RemovedMethods>
    </ModifiedFile>
  </ModifiedFiles>
</SummaryReport>
```

Figure 5.6.: Example of an Overview Report

2. **Bundle Mode:** compares two interface packages (.zip) of the Platform, which are locally stored.
3. **Artifactory Mode:** compares two interface packages (.zip) of the Platform, which are available on Artifactory.

Single Mode

Creates report files for single library file pairs, optional an ignore list can be provided.

Usage from the Command Line:

```
ApiDiff.exe -s "<path to first library file>" "<path to the second library file>"
[-i "<path to alternative ignore list>\IgnoreList.txt"] -rd "<path to a directory
for storing the results>"
```

Example:

```
..\..\ApiDiff.exe -s "..\ApiDiff_Test\Testdata\SingleDLL\1.2-790.0\firstLibraryFile.dll"
..\..\ApiDiff_Test\Testdata\SingleDLL\1.2-809.0\secondLibraryFile.dll" -rd "..\..\ApiDiff"
```


Bundle Mode

Creates report files according to interface packages, which are stored locally. Paths to the two different interface packages have to be provided. Optional the name of a specific product can be provided with paths for the corresponding usage lists, for generating product specific reports. Another option is the prompt for the ignore list, in that case, classes which are listed in the ignore list, won't be considered. An "usage list" consists of implemented interfaces of the Platform or called methods of the Platform. Those lists are generated by parsing products (Product 1, Product 2), which use functionality of the Platform.

Usage from the Command Line:

```
ApiDiff.exe -b "<path to the first interface package>" "<path to the second interface package>" [-p "<name of the product>" -uc "<path to file>\UL_CalledMethods_<name of the product>.txt" -ui "<path to the file>\UL_ImplementedInterfaces_<name of the product>.txt"'] [-i "<path to alternative ignore list>\IgnoreList.txt"'] -rd "<path to a directory for storing the results>"
```

Example for Creating "Product 2" Specific Reports:

```
..\..\ApiDiff.exe -b "..\ApiDiff.Test\Testdata\Packages\Platform-1.2-790.0-Interfaces.zip"
..\ApiDiff.Test\Testdata\Packages\Platform-1.2-820.0-Interfaces.zip" -p "Product
2" -uc "..\UsageLists\UL_CalledMethods_Product2.txt" -ui "..\UsageLists\
UL_ImplementedInterfaces_Product2.txt" -i "..\IgnoreLists\IgnoreList.txt" -rd
..\..\ApiDiff"
```

Artifactory Mode: Individual

Creates report files for interface packages, which are stored on Artifactory. Names of the target repositories, the major number, the corresponding version numbers, the directory in which dependent tools are located for uploading the reports to Artifactory and a directory for storing the results (those results will then be compressed and uploaded to Artifactory) have to be provided. Optional the name of a specific product can be passed, for generating product specific reports. Another option is the prompt for the

5. Tools for Determining API Breaks

ignore list, in that case, classes which are listed in the ignore list, will not be considered.

Usage from the Command Line:

```
ApiDiff.exe -a -r "<name of the first repository>" -R "<name of the second repository>" -v <first version number> -V <second version number> -m <major number> [-p "<name of the product>" -uc "<path to the file >\UL_CalledMethods_<name of the product>.txt" -ui "<path to file>\UL_ImplementedInterfaces_<name of the product>.txt"] [-i "<path to alternative ignore list>\IgnoreList.txt"] -fd "<path to the 'Tools' directory>" -rd "<path to a directory for storing the results>"
```

Example for Creating "Product 1" Specific Reports:

```
..\..\ApiDiff.exe -a -r "repo-qa" -R "repo-snapshot" -v 922.0 -V 954.0 -m 1.2 -p "Product1" -uc "..\UsageLists\UL_CalledMethods_Product1.txt" -ui "..\UsageLists\UL_ImplementedInterfaces_Product1.txt" -i "..\IgnoreLists\IgnoreList.txt" -fd "....\..\Tools" -rd "....\ApiDiff"
```

Artifactory Mode: Default

Creates report files for interface packages, which are stored on Artifactory. Names of the target repositories, the major number, the directory in which dependent tools are located for uploading the reports to Artifactory and a directory for storing the results (those results will then be compressed and uploaded to Artifactory) have to be provided. In that case, the current interface packages of the provided repositories will be taken. Optional the name of a specific product can be provided, for generating product specific reports. Another option is the prompt for the ignore list, in that case, classes which are listed in the ignore list, will not be considered. In contrast to the individual mode, the default mode performs a comparison of the latest interface packages of the provided repositories.

Usage for the Command Line:

```
ApiDiff.exe -a -r "<name of the first repository>" -R "<name of the second repository>" major number [-p "<name of the product>" -uc "<path to the file >\UL_CalledMethods_<name of the product>.txt" -ui "<path to
```

5.1. ApiDiff

```
file>\UL_ImplementedInterfaces_<name of the product>.txt") [-i "<path to  
alternative ignore list>\IgnoreList.txt"] -fd "<path to the 'Tools' directory>"  
-rd "<path to a directory for storing the results>"
```

Example for Creating Common Reports for Major Number 1.1:

```
..\..\ApiDiff.exe -a -r "repo-qa" -R "repo-snapshot" -m 1.1 -fd "..\..\Tools" -rd  
..\..\ApiDiff"
```

Example for Creating "Product 1" Specific Reports for Major Number 1.2

```
..\..\ApiDiff.exe -a -r "repo-qa" -R "repo-snapshot" -m 1.2 -p "Product 1" -uc  
..\UsageLists\UL_CalledMethods_Product1.txt" -ui "..\UsageLists\  
UL_ImplementedInterfaces_Product1.txt" -i "..\IgnoreLists\IgnoreList.txt" -fd "..\..\Tools"  
-rd "..\..\ApiDiff"
```

5.1.6. Directory Hierarchy

By using ApiDiff, a directory hierarchy e.g.: in figure 5.7 will be created locally (not in Artifactory) for storing different files. Those folders are necessary for storing downloaded interface packages (zip compressed), extracted interface packages and the generated report files (xml files).

5.1.7. Testing

ApiDiff is tested by several unit tests and functional tests, which are integrated and applied by a TeamCity server. All tests, which are integrated into a Continuous Integration server- as shown in figure 5.8 will at first check out the latest version of ApiDiff from a specific branch. After that, ApiDiff will be built and the tests start.

5. Tools for Determining API Breaks

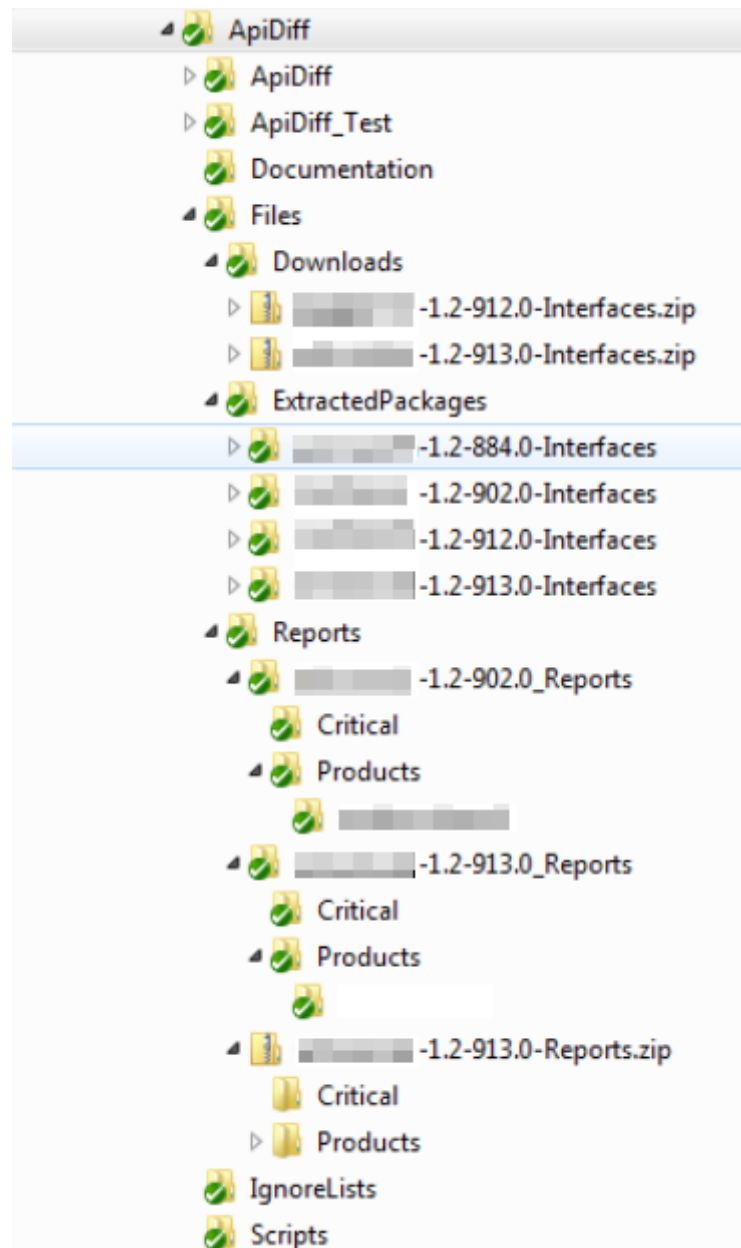
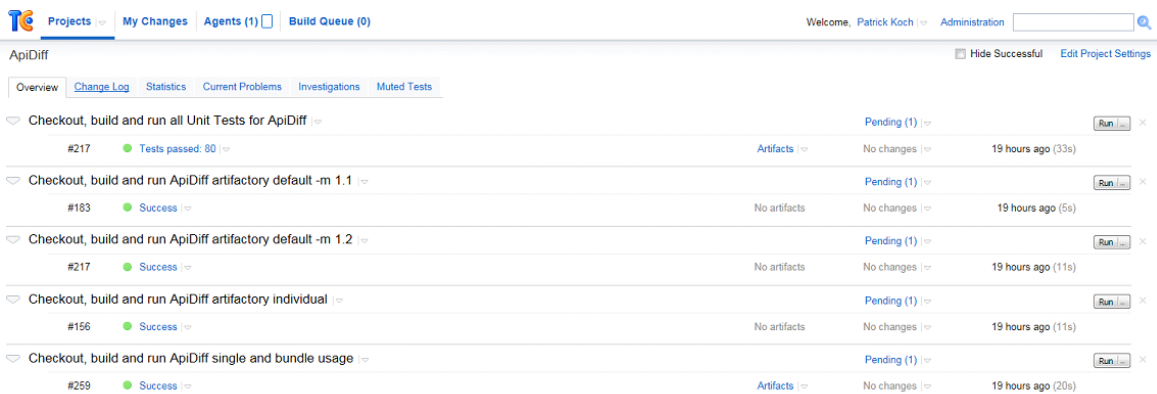


Figure 5.7.: Directory hierarchy overview

5.1. ApiDiff



Build Type	Status	Artifacts	Changes	Time
Checkout, build and run all Unit Tests for ApiDiff	Pending (1)	Artifacts	No changes	19 hours ago (33s)
Checkout, build and run ApiDiff artifactory default -m 1.1	Pending (1)	No artifacts	No changes	19 hours ago (5s)
Checkout, build and run ApiDiff artifactory default -m 1.2	Pending (1)	No artifacts	No changes	19 hours ago (11s)
Checkout, build and run ApiDiff artifactory individual	Pending (1)	No artifacts	No changes	19 hours ago (11s)
Checkout, build and run ApiDiff single and bundle usage	Pending (1)	Artifacts	No changes	19 hours ago (20s)

Figure 5.8.: Provided tests for ApiDiff, integrated on a TeamCity server

5.1.8. Maintenance

Integration of ApiDiff in a Continuous Integration system

A recommend way for integrating ApiDiff in a Continuous Integration system like TeamCity would be necessary to check it out, build and run it. ApiDiff will be built every time, to avoid to store the executable file in the version control system. In the current situation (as described), there is just one batch file necessary with following content seen in figure 5.9.

```
1 @echo off
2
3 C:
4 cd C:\Windows\Microsoft.NET\Framework64\v4.0.30319\
5
6 MSBuild.exe ".....\Tools\ApiDiff\ApiDiff.sln" /p:Configuration=Release /p:Platform=x86
7
8 D:
9 cd .....Tools\ApiDiff\ApiDiff\bin\Release\
10
11 REM run ApiDiff for ..... specific reports
12 ApiDiff.exe -a -r "repo-qa" -R "repo-snapshot" -m 1.2 -p "....." -uc ".....\UsageLists\UL_CalledMethods_......txt"
13 -ui ".....\UsageLists\UL_ImplementedInterfaces_......txt" -i ".....\IgnoreLists\IgnoreList.txt"
14 -fd ".....\Tools" -rd ".....\ApiDiff"
15
16 REM run ApiDiff for ..... specific reports
17 ApiDiff.exe -a -r "repo-qa" -R "repo-snapshot" -m 1.2 -p "....." -uc ".....\UsageLists\UL_CalledMethods_......txt"
18 -ui ".....\UsageLists\UL_ImplementedInterfaces_......txt" -i ".....\IgnoreLists\IgnoreList.txt"
19 -fd ".....\Tools" -rd ".....\ApiDiff"
```

Figure 5.9.: Batch file, responsible for building and running ApiDiff

5. Tools for Determining API Breaks

This batch file includes commands for building ApiDiff (line 6) and running it (line 11 and line 16). ApiDiff is started in two different ways: in both usages it is going to download the latest interface packages of the QA and Snapshot repository, but in line 11 Product 1 specific reports will be created and in line 16, Product 2 specific reports will be created.

5.1.9. Automated TeamCity Testing

The TeamCity job "Checkout, build and run ApiDiff artifactory individual" examines the individual modes of ApiDiff. For that, specific interface packages will be compared with each other and the corresponding version number has to be provided. Those tests will not work after a specific point of time, because a Nightly Build of the Platform creates new packages every day and increments the version number (which is equal to the build number). Older interface packages are removed after a while. This job has to be kept current by updating the version numbers.

5.1.10. Future Work

Improvements according to functionality

Recognition of Pure Virtual Methods:

ApiDiff detects added/removed classes and added/removed methods. For the creation of critical reports, added types will not be considered. An exception exists in this rule: Adding a pure virtual method in an abstract base class would lead to an interface break too because the client software needs to implement a pure virtual method. So this special case should be considered.

More Precise Matching of Method Signatures:

Differences between methods are recognized by comparing the method signatures. Mono.Cecil, which is used for parsing library files, provides functionality for that way. It works fine at the time, but using Mono.Cecil functions would be a much cleaner way.

5.1.11. Improvements according to Tests

Automated TeamCity Tests:

The TeamCity test "Checkout, build and run ApiDiff artifactory individual" examines the individual modes of ApiDiff. For that, specific interface packages will be compared with each other and the corresponding identifier has to be provided. Those tests will not work after a specific point of time, because a Nightly Build of the platform creates new packages every day and increments the version number. Older interface packages are removed after a while. Keep it current.

Unit Tests:

The following unit test, as seen in figure 5.10 tries to test the download functionality of ApiDiff. For that, the latest interface package with the major number 1.2 will be downloaded. At some time, the major number 1.2 will not be current any more- so this number has to be increased to 1.3.

5.2. AssemblyParser

5.2.1. Motivation

The aim of this tool is providing so-called "usage lists" for ApiDiff. An usage list consists of classes, interfaces and methods of the Platform which are referenced in different ways in the Platform dependent products: e.g. Product 1 or Product 2. ApiDiff uses those usage lists for filtering for product specific changes according to releases of the Platform: for instance which changes in the new version of the Platform have an influence to Product 1 or Product 2?

5. Tools for Determining API Breaks

```
[TestMethod]
[DeploymentItem("Testdata", "Testdata")]
public void TestDownload(InterfacePackage_MajorNumber1Dot2()
{
    // You may have to change the Major Number, e.g. if "1.2" isn't available any more

    // arrange
    string fileName = null;
    string pathForDownloadedFile = null;
    string versionNumber = null;

    cmdLineParser = new CommandLineParser();
    artifactoryFetcher = new ArtifactoryFetcher();

    fileName = artifactoryFetcher.GetFileNameOfLatestVersion("repo-snapshot", "1.2");
    versionNumber = ConventionChecker.Instance.GetVersionNumberOfPackage(fileName);
    pathForDownloadedFile = string.Concat(pathForDownload, @"\", fileName);

    // act
    artifactoryFetcher.download(InterfacePackage("repo-snapshot", "1.2", versionNumber, pathForDownload)

    // assert
    Assert.IsTrue(File.Exists(pathForDownloadedFile));
}
```

Figure 5.10.: Unit Test of ApiDiff

5.2.2. Usage of AssemblyParser

AssemblyParser needs as input among others a directory, which contains library files. Those library files will be parsed with help of Mono.Cecil, for figuring out which specific classes, interfaces and methods of the Platform are referenced.

AssemblyParser.exe -p "<Name of the Product>" **-d** "<Path to the Library Files>" **-w** "<path to a working directory, for storing the results >" [**-f** "*Additional Filter for Files*.dll"]

- **-p**: Name of the product, e.g.: "Product 1"
- **-d**: Path to the directory, which contains the library files (.dll)
- **-w**: Path to a working directory, (e.g.: a random folder). In that folder the result files (usage lists) will be generated.
- **-f**: Optional filter, at first only files which matches with the product name will be parsed. Furthermore, a filter can be provided for parsing

5.2. AssemblyParser

additional files. This could be helpful if you do want to match files too which are not named after the product.

If there is no filter provided, the tool will scan files, which contain the product name.

In the following lines, examples for applying AssemblyParser will be provided.

Instructions for Product 1 without Filtering

Applying the tool for parsing for "Product 1" library files. The passed path to the directory will be scanned for file names, which consist "Product 1".

```
AssemblyParser.exe -p "Product 1" -d "\\nbuild11\Platform_V1\Subsystems  
\Product1\Bin\Release" -w "..\..\WorkingDirectory"
```

Instructions for Product 1 with Filtering

In this example, a filter is provided. In that case the library files are filtered according to the filter, in that specific case every file which includes "Product 1" or "Common" will be parsed.

```
AssemblyParser.exe -p "Product 1" -d "\\nbuild11\Platform_V1\Subsystems  
\Product1\Bin\Release" -w "..\..\WorkingDirectory" -f "*Product 1*.dll  
*Common*.dll"
```

5.2.3. Integration of AssemblyParser in a Continuous Integration System

There is very little effort for integrating AssemblyParser to TeamCity: only three build steps are necessary as seen in figure 5.11:

- Building AssemblyParser
- Creating of a folder, which serves as working directory for storing the results

5. Tools for Determining API Breaks

- Executing AssemblyParser

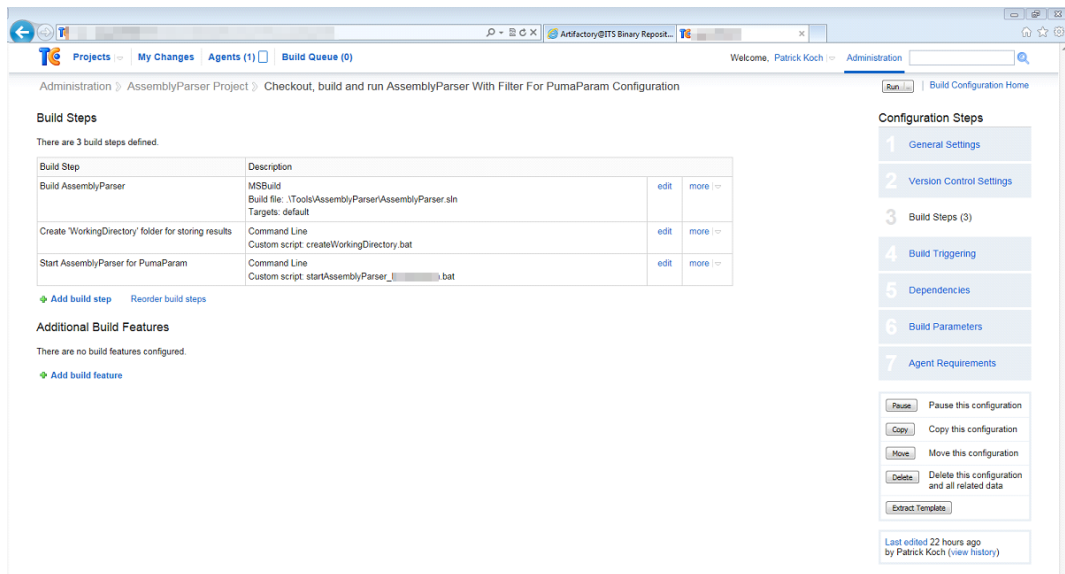


Figure 5.11.: Integration of AssemblyParser in TeamCity

The referenced batch script 5.12 of the second build step consists just of two commands:



Figure 5.12.: Batch script for starting AssemblyParser

5.2.4. Future Work for AssemblyParser

The collaboration with ApiDiff is not fully automated at the moment. The passed paths, which reference the directories of the stored library files have to be adapted manually, if the location of the storage changes. The implementation of a fully automated retrieving of those library files would improve the automation of the whole process.

5.3. Conclusion

The tools presented in this chapter were developed for determining API changes of the Platform and therefore for preventing build breaks. Every developer of the Platform is able to use this functionalities. Before someone is going to release specific changes to the platform, it is possible to apply ApiDiff to the current version and a quality assured one, for determining API changes. If there are no critical changes, the current changes of the Platform can be released. As a post control, ApiDiff will generate daily reports of the current versions of the snapshot and the quality assured repositories. This should serve as a contribution for ensuring the quality of the Platform.

6. Summary

API management is a very important topic to any software developer. There is a seemingly endless list of good practices, how a good API development can be achieved. Beside the knowledge about good guidelines for achieving high quality APIs, suitable development approaches will improve APIs too. I want to give a summary about the whole work, by providing a recap about every chapter. The first chapter deals with API management: how can we develop and maintain high quality APIs? There exist a lot of good practices for achieving that, but I think the most important questions, every developer of an API should be faced with are: "what does the customer expect?", "how can we make life for the customer easier?", "how can backward compatibility be guaranteed?" and "how can we maintain the API that no customer will take notice about it?". Those questions should always be taken into account, when facing API aspects. The second chapter was about agile software development. Small iterative processes may probably have a positive impact on API management, because the aim of every cycle is to provide a working software and the goal of every cycle will be determined at the beginning. Therefore, there is no uncoordinated work possible. Agile approaches set also a great value on communication, which is a very important aspect for API management too, because it is all about the customer, who will use the API. The third chapter consists of the results of an expert interview, which I have performed with experienced developers. I confronted them with questions about API development and agile approaches. It transpired that they agree with most the guidelines, which I have asked them about. In general they all have a good opinion about agile approaches too but they consider it critically. The next to last one tries to find synergies between agile approaches and API development. Of course, there are many, because API development differs with regard to a common software project in one critical issue: if the API is released, then the interface contract has to be kept for the whole durability of the API. Agile approaches should support the

6. Summary

API development more intensive, especially before the release date. The last chapter is about two C# applications, which provide rapid feedback in the case of possible API violations. It is obvious that no company can achieve a perfect API development, therefore preventive measures have to be taken.

Experience and theoretical knowledge about best practices in API development are definitively important factors for achieving high quality APIs, but combined with the concepts of agile software development approaches the development can be optimized.

Appendix A.

Interview Guide

To facilitate matters, the full questionnaire will be provided and enumerated to assign the answer of each participant as easy as possible.

Part 1

- **a:** In your opinion from the developer's point of view, what are the advantages of systems, which are using APIs in contrast to systems, which do not?
- **b:** Have you ever participated in the development of an API?
- **c:** Have you ever faced a situation, in which an adaptation of an API would make sense?

Part 2

- **a:** Please move back to a situation, in which you had to try an API. You got, what you have expected, the API worked well. Please describe the properties of this specific API, respectively- in your opinion, what are the qualities of a well designed API?
 - **b:** Please move back to a situation, in which you had to try an API. In this case you did not get, what you have expected, the API worked not well. Please describe the properties of this specific API, respectively- in your opinion, what are the qualities of a bad designed API?
1. **c:** Do not expose member variables, which are not part of the interface
 2. **d:** Develop an API as a software system with a low coupling and a high cohesion

Appendix A. Interview Guide

3. **e:** An API should be self-explanatory, no documentation should be necessary
4. **f:** If a redesign of the architecture would need too much effort, try to implement a wrapping of the interface by using a Proxy, Adapter or Facade pattern
5. **g:** API testing should be achieved with Unit and Integration tests

Part 3

- **a:** Do you see any risks by applying the five best practices?
- **b:** Do you think, that diverse problems could be avoided by applying agile software development? If yes, which one?

Appendix B.

Interview Partners

This sequence of interview partner is literal. The usage of the pseudonymisation codes in the text was randomly performed. I have chosen those specific interviewees, because of their experience in software development and their responsible position at AVL. They all have often faced situations, related with API management- therefore they were predestined for this expert interview.

- MMag. Igor Roncevic, AVL List GmbH (Project Leader Development Project)
- Dipl.Ing. Stefan Preuer, AVL List GmbH (Software Architect)
- Dipl.Ing. Astrid Lock, AVL List GmbH (Department Manager and former Project Leader Development Project)
- Dr. Harald Rosenberger, AVL List GmbH (Project Leader Development Project)
- Dipl.Ing. Andreas Fischer, AVL List GmbH (Department Manager)

Appendix C.

Examples

C.1. Examples of Information Hiding

Listing C.1: Examples of declarations in C++

```
void RandomMethod(int firstParameter);  
int counter;  
class Person;
```

Listing C.2: Examples of definitions in C++

```
void RandomMethod(int firstParameter)  
{  
    printf("Passed argument was: %d\n", firstParameter);  
}  
  
int counter=10;  
  
class Person  
{  
public:  
    char* forename, lastname;  
};
```

Appendix C. Examples

Listing C.3: Examples of definitions in C++ header files

```
// Person.h

class Person
{
    public:
    void printName()
    {
        printf("Providing a definition in a header file.\n");
    }
};
```

Listing C.4: Class with public members

```
// Person.h

class Person
{
    public:
    char* name;
    int age;
};
```

Listing C.5: Class with implemented getter and setter methods

```
// Person.h

class Person
{
    public:
    char* GetName();
    int GetAge();
    void SetName(char* val);
    void SetAge(int val);

    private:
    char* name;
    int age;
};
```

C.1. Examples of Information Hiding

Listing C.6: Example of a stack with public members

```
// Stack.h

Class Stack
{
public:
static const int MAX_SIZE = 10;
void Push(int value);
int Pop();
int theStack[MAX_SIZE];
int topOfTheStack();
int currentSize;
};
```

Listing C.7: Example of a stack with private members

```
// Stack.h

Class Stack
{
public:
void Push(int value);
int Pop();
int topOfTheStack();

private:
int currentSize;
int theStack[MAX_SIZE];
static const int MAX_SIZE = 10;
};
```

Listing C.8: Certificate example

```
// RetrieveCertificatesOfX509Store.h

class RetrieveCertificatesOfX509Store
{
public:
```

Appendix C. Examples

```
RetrieveCertificatesOfX509Store();

StoreName GetStoreName() const;
StoreLocation GetStoreLocation() const;

X509Store CreateAndOpenStore(StoreName storeName,
StoreLocation storeLocation);
bool CloseStore(X509Store store);
bool SetStoreName(StoreName storeName);

bool RetrieveCertificates(X509Store store);

private:
X509Store store;
StoreName storeName;
};
```

Listing C.9: The class 'PictureShow' needs the usage of the class 'SinglePicture', which is totally set to private

```
// PictureShow.h

class PictureShow
{
public:
PictureShow();
void SetSpeed(int speed);
void SetNumberOfPictures(int num);
void StartShow();
void Break();
void NextPicture();

private:
class SinglePicture
{
public:
float posX, posY;
int speedX, speedY;
```

C.2. Examples of Aspects of Coupling

```
int appearance;
};
double posX, posY;
float speed;
bool isVisible;
std::list<SinglePicture *> singlePicturesList;
};
```

C.2. Examples of Aspects of Coupling

Listing C.10: Forward declaration of a class in C++

```
// ClassTwo.h

class ClassOne; // forward declaration

class ClassTwo
{
public:

ClassTwo();
void SetClassOneObject(ClassOne *classOneObjectInstance);
ClassOne *GetClassOneObject() const;

private:
ClassOne *classOneObject;
};
```

Listing C.11: Class with member functions only

```
// MyClass.h

class MyClass
{
public:
void PrintAddress() const;
void SetAddress(char* addressValue);
std::string GetAddress() const;
```

Appendix C. Examples

```
private:
char* address;
};
```

Listing C.12: Class with outsourced function

```
// MyClass.h

class MyClass
{
public:

void SetAddress(char* addressValue);
std::string GetAddress() const;

private:
char* address;
};

void PrintAddress(MyClass &myClassInstance);
```

Listing C.13: A coupling exists between 'Product' and 'ShoppingBasket'

```
#include "Product.h"

class ShoppingBasket
{
public:
bool AddProductToBasket(const Product &product, int amount);
int GetAmount(int index);

private:
struct Purchase
{
Product product;
int amount;
};
```


C.3. Examples of a Singleton Pattern

```
std::vector<Purchase> purchases;
```

Listing C.14: Coupling is solved, but redundant code has to be accepted for that

```
class ShoppingBasket
{
public:
    bool AddProductToBasket(char* nameOfProduct, int amount);
    int GetAmount(int index);

private:
    struct Purchase
    {
        char* nameOfProduct;
        int amount;
    };

    std::vector<Purchase> purchases;
```

C.3. Examples of a Singleton Pattern

Listing C.15: Example implementation of a Singleton class

```
// MySingleton.h

class MySingleton
{
public:
    static MySingleton &GetInstance();

private:
    MySingleton();
    ~MySingleton();
    MySingleton(const MySingleton &);
    const MySingleton &operator =(const MySingleton &);
};
```

Appendix C. Examples

Listing C.16: Initialization of a Singleton

```
// MySingleton.cpp

MySingleton &MySingleton::GetInstance()
{
    static MySingleton instance;
    return instance;
}
```

Listing C.17: Simple C++ class without using a dependency injection

```
// ExampleClass.h

class ExampleClass
{
    ExampleClass() :
        myLogger(new Logger("mylogger", "defaultformat"))
    {}

private:
    Logger *myLogger;
};
```

Listing C.18: Simple C++ class with using a dependency injection

```
// ExampleClass.h

class ExampleClass
{
    ExampleClass(Logger *log) :
        myLogger(log)
    {}

private:
    Logger *myLogger;
};
```

C.4. Examples of a Factory Pattern

C.4. Examples of a Factory Pattern

Listing C.19: Simple factory pattern example in C++

```
// FurnitureFactory.h

// for including "Furniture" class, which is an abstract base
class
#include "Furniture.h"
#include <string>

class FurnitureFactory
{
public:
Furniture *GenerateFurniture(const std::string &typeOfFurniture)
    ;
};
```

Listing C.20: Simple factory pattern implementation example in C++

```
// FurnitureFactory.cpp

#include "FurnitureFactory.h"
#include "Desk.h"
#include "Chair.h"
#include "Cupboard.h"

Furniture *FurnitureFactory::GenerateFurniture(
const std::string &typeOfFurniture)
{
if (type == "desk")
    return new Desk();
if (type == "chair")
    return new Chair();
if (type == "cupboard")
    return new Cupboard();

return NULL;
}
```

C.5. Examples of a Proxy Pattern

Listing C.21: Proxy pattern implementation example in C++

```
// ProxyClass.h

class ProxyClass
{
public:
ProxyClass() : realObject(new RealObject())
{}
~ProxyClass()
{
delete realObject;
}
bool DoSomething(int value)
{
return realObject->FunctionOne(value);
}
private:
ProxyClass(const ProxyClass &);
const ProxyClass &operator =(const ProxyClass &);
RealObject *realObject;
};
```

C.6. Examples of an Adapter Pattern

Listing C.22: Adapter pattern implementation example in C++

```
// SquareAdapter.h

class SquareAdapter
{
public:
SquareAdapter() :
    mySquare(new Square())
{}
};
```

C.6. Examples of an Adapter Pattern

```
~SquareAdapter()
{
    delete mySquare;
}
void SetSquare(float x1, float y1, float x2, float y2)
{
    float length = x2 - x1;
    mySquare->setSquareObject(x1, y1, length);
}
private:
SquareAdapter(const SquareAdapter &);
const SquareAdapter &operator =(const SquareAdapter &);
Square *mySquare;
};
```


References

- Apke, L. (2014, February). *The agile principles: Face to face conversation - larry apke*. <http://www.agile-doctor.com/2014/02/11/face-face-conversation/>. ([Online; accessed 20-September-2014])
- Beck, K. (2001). *Principles behind the agile manifesto*. <http://www.agilemanifesto.org/principles.html>. ([Online; accessed 19-September-2014])
- Bloch, J. (2007, January). *Api design and development guidelines - selected resources - 3scale - the api management solution*. <http://www.3scale.net/2010/09/api-design-development-selected-resources/>. ([Online; accessed 22-November-2014])
- Henning, M. (2009, May). *Api design matters — may 2009— communications of the acm*. <http://cacm.acm.org/magazines/2009/5/24646-api-design-matters/fulltext>. ([Online; accessed 18-September-2014])
- Huston, T. (n.d.). *What is agile testing?* <http://smartbear.com/products/qa-tools/what-is-agile-testing/>. ([Online; accessed 07-November-2014])
- Jeffries, R. E. (n.d.). *What is extreme programming? — xprogramming.com*. <http://xprogramming.com/what-is-extreme-programming/>. ([Online; accessed 03-November-2014])
- Jenkov, J. (n.d.). *Understanding dependencies*. <http://tutorials.jenkov.com/ood/understanding-dependencies.html>. ([Online; accessed 10-November-2014])
- Ken Schwaber, J. S. (2013, July). *The scrum guide*. Retrieved from <http://www.scrumguides.org/> ([Online; accessed 31-December-2014])

References

- Lotz, M. (2013, July). *Waterfall vs. agile*. <http://www.seguetech.com/blog/2013/07/05/waterfall-vs-agile-right-development-methodology>. ([Online; accessed 29-December-2014])
- Mieg, H. (2005, April). *Experteninterviews*. <http://www.mieg.ethz.ch/education>. ([Online; accessed 10-November-2014])
- Pedro, B. (2014, October). *5 reasons why developers are not using your api*. <http://nordicapis.com/5-reasons-why-developers-are-not-using-your-api/>. ([Online; accessed 12-November-2014])
- Preston-Werner, T. (n.d.). <http://semver.org/> — *semantic versioning 2.0.0*. <http://semver.org/>. ([Online; accessed 04-September-2014])
- Purdy, D. (2002, February). *Exploring the factory design pattern*. <http://msdn.microsoft.com/en-us/library/ee817667.aspx>. ([Online; accessed 07-January-2015])
- Reddy, M. (2011). *Api design for c++*. Elsevier.
- Robinson, S., Nagel, C., Watson, K., Glynn, J., Skinner, M., & Evjen, B. (2004). *Professional c# 3rd edition* (3rd ed.). wrox.
- Shoukath, N. (2012, July). *Continuous delivery - continuous deployment explained — people10 - youtube*. <http://www.youtube.com/watch?v=gH-fTlitrok>. ([Online; accessed 08-December-2014])
- Smolaks, M. (2012, July). *Agile development comes with hidden costs — techweekeurope uk*. <http://www.techweekeurope.co.uk/news/agile-development-report-86052>. ([Online; accessed 03-November-2014])
- Sugrue, J. (n.d.). *Design patterns uncovered: The adapter pattern — java lobby*. <http://java.dzone.com/articles/design-patterns-uncovered-o>. ([Online; accessed 06-January-2014])
- Tilloo, R. (2014, October). *Waterfall modell in software engineering- technotrice*. <http://www.technotrice.com/what-is-waterfall-model-software-engineering/>. ([Online; accessed 31-October-2014])

References

- Townsend, M. (2002, February). *Exploring the singleton design pattern*. <http://msdn.microsoft.com/en-us/library/ee817670.aspx>. ([Online; accessed 07-January-2015])
- Tulach, J. (2008). *Practical api design*. Apress.
- Waters, K. (2007, June). *10 good reasons to do agile development — all about agile*. <http://www.allaboutagile.com/10-good-reasons-to-do-agile-development/>. ([Online; accessed 07-January-2015])
- Zolyak, A. (2013, April). *8 benefits of agile software development*. <http://www.seguetech.com/blog/2013/04/12/8-benefits-of-agile-software-development>. ([Online; accessed 31-December-2014])