Agile Development with Stepwise Feature Introduction

Mikołaj Olszewski[1,2] and Ralph-Johan Back[1]
[1]*Department Of Information Technologies, Åbo Akademi University, Joukahaisenkatu 3-5, Turku, Finland*
[2]*Turku Centre for Computer Science (TUCS), Graduate School, Turku, Finland*

Keywords: Agile Development, Object-oriented Design, Development Paradigm, Open-ended Development.

Abstract: The paper evaluates the Stepwise Feature Introduction paradigm, an organised method for constructing layered, reusable, object-oriented software systems. Based on our research adapted the paradigm to construction of large-scale software systems. In particular, we added a dedicated, agile development process to the paradigm and examined strategies for execution and testing. Correctness concerns of the produced system are also covered in this paper. We also briefly analyse the impact of the paradigm on the quality of the developed software.

1 INTRODUCTION

The paradigm of Stepwise Feature Introduction (SFI) has been developed by R.-J. Back as a high-level framework for software development (Back, 2002). SFI facilitates the construction and evolution of a software system by introducing features incrementally, one after another, in a layered manner.

SFI has been applied to the development of a number of proof-of-concept software, e.g. a calendar application (Back, et al., 2005). These projects have confirmed that the gradual extension of functionality, which is the essence of the paradigm, promotes high reusability and maintainability of the constructed software. Recently the paradigm was used for construction and reengineering of more complex software systems. During that work the paradigm evolved from theoretical grounds to a fully customisable agile framework.

This paper presents the current state of the paradigm and is organised as follows. We start with the introduction of the paradigm and its basic concepts in section 2. Section 3 discusses the development process. Correctness and testing are presented in section 4, together with diagrammatic reasoning. The discussion on the quality follows in section 5. We conclude our presentation of the paradigm with the overview of alternative approaches to software construction, as well as with some general remarks and directions for our future work. An extended version of this paper, where a specific case study is used to illustrate the concepts involved, is available as a Turku Centre for Computer Science Technical Report (Olszewski & Back, 2012).

2 OVERVIEW

There are three important characteristics of a system created with SFI that can be outlined. First, the paradigm requires that whenever new functionality is added, the preservation of old features must be explicitly checked. Second, the system in construction must be fully executable after each added feature. And finally, a working version must be presented to its users and stakeholders once new features are added.

The paradigm requires a programming language that supports subtype polymorphism and inheritance. The former is required in order for the new features to be used in the context of the old ones, although it does not guarantee that the functionality is preserved. Inheritance, on the other hand, enables to extend an existing feature while maintaining parts of its original behaviour. Thus, object-oriented programming languages are a natural choice for SFI.

As the system grows, new features are added to it in the form of layers, one after another. The new functionality may extend or utilise services provided in the earlier layers. It is also possible to combine, replace, rearrange and remove layers, for example, to enable more efficient algorithms or to optimise

Olszewski M. and Back R..
Agile Development with Stepwise Feature Introduction.
DOI: 10.5220/0003997901610166
In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2012), pages 161-166
ISBN: 978-989-8565-13-6

the design.

2.1 Service Providers, Users, Borders

The interacting elements of the system – its components – are differentiated based on the role they play. *Service providers*, as the name implies, offer a specific functionality to the other components by directly implementing their functions, without relying on or using the remaining parts of the software.

Service users, on the other hand, utilise the functionality offered by the providers. These components enable the provided services to be effectively used during the operation of the system. They are also the final elements in dependency chains, as no other parts of the system depend on them. In the most common case, however, the elements of the system provide functionality by relying on other components, thus being both service providers and service users. The role of such components depends on the perspective they are examined from.

Fully abstract components may also be found in the design of a system, especially in a large and complex one. Such components play the role of *service borders*, which outline the desired behaviour for their descendants.

2.2 System Execution

We mentioned previously that a system developed with SFI must be executable at each layer. In other words, the layer and all its preceding layers must constitute a working, executable system. The paradigm provides a number of ways in which this property can be satisfied, based on their complexity.

Executable method is the most straightforward solution, applicable to the simplest cases. It requires that each layer contains service user(s) with dedicated method(s) for executing the system at that particular layer.

Inheritable executable method makes the service user(s) with executable methods parts of the same inheritance tree. While it allows reusing the executable code in the earlier layers, it may also make the rearranging of layers more difficult.

By adding a *dedicated service user* to a layer it is possible to encapsulate the code related to the execution in a separate class. Reusing the layer in a different setting may, however, require additional changes to the dedicated service user.

A *hierarchy of dedicated users* is a natural extension of the previous method. As with inheritable executable method, the major drawback is the limited rearranging of the layers.

Creation of a *dedicated system executable* is the most complex, yet the most flexible option and thus suitable for large and complex systems. In this case a configuration file or command line parameters are used to identify the layer that the execution should start with.

Each increment in functionality produces a new executable version of the system, although possibly with limited set of features. This characteristic of the paradigm results in a collection of systems being built, rather than a single system. Therefore, it is possible to remove a number of layers from an existing system and continue the development in another direction, with a distinct set of features.

2.3 Requirements, Components, Layers

The development of a software system starts by specifying its desired behaviour, based on market research, interaction with potential users, etc. These goals set for the development are the *requirements* of the final system, i.e. its high-level features.

The requirements, although well-defined, are abstract and can be implemented in various ways. It is usually the decision of the system architects and designers to analyse the requirements and identify the main *components* of the software. It is by no means necessary to establish the components that realise all of the requirements; a more advisable solution is to concentrate on the basic functionality first and extend the components later. The knowledge about all or most of the requirements, however, can significantly improve the design and its future modifications.

As the development progresses, the classes that contain code and implement the interfaces are gradually introduced. Each class delivers a small increment to the functionality of the component it belongs to.

The paradigm explicitly states that the software must be executable once new behaviour was added. Therefore, new functionality added to a service provider must be accompanied with a code that utilises it, i.e. corresponding service users. The related classes are thus added to the system together, in the form of layers.

3 DEVELOPMENT PROCESS

The software built with SFI is open-ended, i.e. it can be extended and modified once it is complete (Back,

2002). Moreover, it is constructed through a number of iterations, each adding a small, but specific and well-defined increment to the functionality. Agile development philosophy shares these characteristics as well (Beck, et al., 2001); therefore an agile process is a natural choice when developing software with SFI. When first presented (Back, 2002), the paradigm recommended the use of Extreme Programming (Beck, 1999). However, in our work we decided to use another agile process, Scrum (Schwaber, 1995) (Schwaber, 2004). It does not provide nor enforce any techniques by itself, rather it serves as a framework, within which different processes and methodologies can be encapsulated.

The main purpose of Scrum is to increase the effectiveness of development practices, so that one can improve upon them while providing a framework for development of complex products (Schwaber & Sutherland, 2010). The people directly and indirectly involved in the project are divided into two disjoint groups, *chickens* and *pigs* (Schwaber, 2004). The former contains those, who are indifferent whether the project fails or not, but are otherwise interested in it. The latter includes all those who are committed to realise the project and are responsible for it (Schwaber & Sutherland, 2010).

Within the latter group additional roles can be identified: the Scrum Master, who maintains the process, the Product Owner representing the stakeholders and final users, and the cross-functional Team who actually does the implementation, design and other product-related analysis (Schwaber, 2004).

Similarly to other agile development methods, Scrum is based on frequent releases of the code over the number of iterations of fixed duration, called *sprints*. The desired functionality of the software in construction is described through user stories, which are held in *product backlog* (Schwaber & Sutherland, 2010).

The functionality to be delivered during a sprint is decided during a Sprint Planning Meeting. The corresponding items from the product backlog are moved to *sprint backlog*. These items cannot be modified for the duration of the sprint.

The integration of SFI with Scrum is straightforward, as the analogies can clearly be seen. The requirements, the components and sometimes also the layers correspond to the items in the backlogs. In particular, the requirements defined by the customer can be represented in terms of user stories. The Team can annotate these stories with information about the components or layers needed during the implementation. Furthermore, an extension of a given component, or its introduction to the system, can also be placed in the backlogs to explicitly mark a design decision to be delivered (Olszewski, 2012).

In order to facilitate the introduction of requirements, components and layers to the system, the Sprint Planning Meeting should be divided into two parts, customer- and architecture-centric. The former enables the Product Owner to prioritise the requirements and the Team to select a number of top-priority items to be delivered, similarly to the idea present in Scrum. The architecture-centric discussion focuses on the components relevant to the implementation of the selected requirements. It provides an opportunity for the Team to establish an initial design for the sprint, incorporate it into already existing architecture and to resolve any ambiguities around the requirements (Olszewski, 2012).

During the sprint the items in the sprint backlog remain fixed, i.e. they cannot be modified and no new items may be added. Due to that stability and the precise meaning the items in the backlog can be used to aid the communication between the developers, in particular during the daily scrum.

The evaluation of functionality delivered during the sprint is carried out at the Sprint Review Meeting. The meeting is divided into three parts, design-, architecture- and customer-centric. The goal of the former two is to evaluate the changes to the layers and the components, respectively. The latter involves interaction with the Product Owner. It is used by the Team to present the working product and gather feedback about its current state.

4 ISSUES OF CORRECTNESS

The paradigm puts a strong emphasis on the correctness, making it an essential concern during the development (Back, 2002). SFI does not require any particular technique for ensuring that the correctness conditions hold. The application of formal methods might be suitable to certain systems, in particular the high-critical ones. In most cases, however, the correctness is ensured through rigorous testing.

The principles of the paradigm state four correctness conditions, all of which must be satisfied for each added layer (Back, 2002): *Internal Consistency* (class invariants must be preserved), *Respect* (classes must adhere to the constraints of other classes), *Preserving Old Features* and

Satisfying Requirements.

4.1 Testing

The paradigm benefits from Test-Driven Development (TDD) (Beck, 2003), a technique frequently implemented in agile development settings. TDD recommends that each fragment of code should have a corresponding unit test designed and written prior to the actual implementation (Beck, 2003). Different strategies in writing tests are applied, depending on a role the tested class plays.

The unit tests are especially beneficial when testing service providers. A carefully designed set of unit tests confirms that a provider class is internally consistent and it provides a proper service. Inheritance allows reusing tests and gives an opportunity to use unit tests for also regression testing. This is beneficial when an existing service provider is extended and must be checked that it preserves the functionality introduced earlier.

The crucial part of testing the service users is to ensure that the user respects the constraints of the service it uses. This can be achieved with integration testing, in which different parts of the system may interact. However, the testing of service users is considerably more demanding. Frequently the purpose of a service user is to utilise the service and present it to the end-user of the system. It is usually achieved through a graphical user interface, which may be difficult to test.

The testing of service borders does not require writing dedicated tests, as the borders are abstract and rarely contain any code. Instead, the correctness conditions for borders are directly ensured while testing the descendant classes.

4.2 Diagram-based Reasoning

The Unified Modelling Language (UML) (Object Management Group, 2010) is used to create a model of the system to be constructed. The most commonly used UML diagrams are class diagrams (Fowler, 2004) that present attributes, operations and relations between different classes of the system. There is a mapping between class diagrams and the code they represent, hence class diagrams can be used to document the developed software.

In order to provide an overall view on the state of correctness at a given stage of the development, SFI introduces diagram-based reasoning. It based on two annotations: a question mark '?' and an exclamation mark '!'. There are three basic constructs of class diagrams that can be annotated: class box, association arrow and inheritance arrow (Back, 2002). The annotating is done simultaneously with the development.

The exclamation mark indicates that the correctness condition associated with the construct is established. Marking the constructs with a question mark signifies that it is not known whether or not the corresponding correctness conditions hold, whereas no symbol means that these conditions were not yet of any concern. It is also allowed for the developers to place additional correctness conditions, if they find it beneficial to the project.

Introducing a new layer to the system raises more concerns over the correctness of classes and associations. More specifically, the fact that the newly added subclasses are internally consistent does not mean that the new associations are correct. In order to assert the correctness of the inheritances the regression tests must be designed. The unit tests, used to state internal consistency of the new implementations, can be modified following the technique described in a previous section of the paper. The same tests can also ensure that the aggregation between the new classes can be considered correct.

Establishing the correctness conditions for classes that are directly related to each other allows also inferring a number of correct associations (Back, 2002). These may be useful when reasoning about the correctness of the code reused in a different context, or when a significant number of new classes are added at the same time.

5 IMPACT ON QUALITY

The application of the paradigm enables production of software of good quality. SFI provides a straightforward framework for gradual extension of software systems. This, in turn, enables us to control the complexity and the design of the system, so that at each point of its development it suits best the current needs.

In order to be able to accurately evaluate the quality of the produced system and the impact of the paradigm on the development, we benefit from the principles of Empirical Research (Kitchenham, et al., 2001). This approach is based on experimentation, observation and collecting evidence that confirms or rejects the research hypothesis. Empirical Research is focused on identification, investigation, authentication and progress of theoretical concepts.

We expect the system developed with SFI to be

reusable due to each layer being executable and *maintainable* because of small, manageable increments in functionality. In addition, our expectation is for the system to be highly modular, due to its division to components and layers.

We define maintainability according to the standard ISO/IEC 9126-1 *Software engineering – Product quality* (ISO, 2001) as a characteristic that allows concluding about the degree in which the software can be maintained. Reusability, on the other hand, is understood as ability of software for integration in other systems (ISO, 2001).

The area of metrics dedicated for agile approaches, including Scrum, is young and not well investigated yet (Elssamadisy, 2007). Some work in the domain of quality metrics has been done for lean approaches (Petersen & Wohlin, 2010). However, in order to be able to quantitatively evaluate the impact of our development approach, we relied on well established and empirically validated metrics that are dedicated for object-oriented systems.

The overall quality of a software system is a combination of the quality of design, architecture, code and tests. These areas can be evaluated in a variety of ways, the most common being automatically collected metrics and measurements.

Our assessment is based on the set of Chidamber and Kemerer metrics (Chidamber & Kemerer, 1994). Their analysis provides an insight on the complexity, maintainability and understandability of the system. The set consists of six metrics which can be collected automatically for any class, without detailed investigation of the code: Weighted Methods Count (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response for a Class (RFC) and Lack of Cohesion in Methods (LCOM).

In general, high value of these metrics denotes the increase of the density of bugs and thus the decrease of the overall quality. An exception to that observation is the NOC metric, which is usually positively related with the quality. The optimal values and their thresholds for each metric are often specific to the project the metrics are applied to.

We have applied the paradigm to the development of two software projects of significant complexity and size – a multi-platform board game (Olszewski & Back, 2012) and a highly-specialised tool for analysis and processing of digital microscope images (Olszewski, 2012).

Our results show that, according to the metrics both systems are highly maintainable and reusable. Low values of LCOM indicate highly cohesive systems, which signify the code that is easy to maintain. Likewise, reasonable values of CBO and WMC are a sign of systems that can be easily managed and reused in different settings. In addition our findings were confirmed by the subjective perception of the developers of one of the systems (Olszewski, 2012).

6 ALTERNATIVE PARADIGMS

Maintainable and reusable software can be developed using various other paradigms. Software construction and design, in general, is about recognising components of the system and establishing the connection between them, based on the requirements.

Certain complex functionality, however, may span over a number of components. Aspect-Oriented Development (Kiczales, et al., 1997) is the approach, in which the design is primarily focused on identifying and representing such cross-cutting concepts – aspects. The behaviour brought to the system by aspects is combined by join points during execution time with the static code of components.

Aspect-Oriented Development is based on a modularisation scheme that is different from the one present in SFI. As a result, static domain knowledge is separated from dynamic, frequently changing requirements. The overall maintainability and reusability of the system is thus increased. The drawback of the approach, however, is the necessity of using a dedicated programming language that is able to describe and combine aspects.

An emerging paradigm of Data, Context and Interaction (Reenskaug & Coplien, 2009) is designed specifically for object-oriented systems and therefore can be used with existing tools. Its general goal is the same as the one of Aspect-Oriented Development – to separate non-changing elements of the system (Data) from the dynamic ones (Context and Interaction).

The dissonance between the static code structure and its run-time representation can be observed in object-oriented systems (Gamma, et al., 1994). Data, Context and Interactions aims to minimise this gap by unifying common practices of object-oriented design and by decomposing the system into different perspectives (Reenskaug & Coplien, 2009).

The dynamic parts of the system are injected to the static objects at run-time, therefore Data, Context and Interaction is suitable for modern, dynamic, object-oriented languages. However, not all object-oriented programming languages offer support for such operations. The paradigm of SFI, in contrast,

relies only on inheritance and subtype polymorphism, and therefore fits all object-oriented programming languages.

7 CONCLUSIONS AND FUTURE WORK

This paper presented a novel approach to software construction, the paradigm of SFI. We described the principles of the paradigm and its agile development process based on Scrum.

The overall results of applying SFI to software development are encouraging. The quality measurement results confirmed that the systems constructed according to the paradigm are maintainable and reusable, and thus present the characteristics which are desirable in software development.

At its current stage, however, the suitability of our approach to various types of developments is not yet statistically validated. Therefore, our intent is to apply SFI to a larger number of projects. The development of database- and web-applications is especially in our focus, due to the success of Web 2.0 and rich internet applications.

The applicability of the paradigm to software product lines is also in the scope of our research. We are confident that the ability of the software to be executed after each iteration, combined with its modular architecture, can be beneficial in such setting.

Finally, we would like to investigate the suitability of the paradigm in combination with a different development process and examine it in more formal environments. Such experiments would allow us to identify potential improvements to the paradigm before it can be applied to construction of systems of higher criticality.

REFERENCES

Back, R.-J., Eriksson, J. & Milovanov, L., 2005. *Using stepwise feature introduction in practice: an experience report.* Springer.

Back, R.-J., 2002. Software Construction by Stepwise Feature Introduction. In: *ZB 02: Proceedings of the 2nd International Conference of B and Z Users of Formal Specification and Development in Z and B.* Springer-Verlag.

Beck, K., 1999. *Extreme Programming Explained.* Addison-Wesley.

Beck, K., 2003. *Test-Driven Development by Example.* Addison Wesley.

Beck, K. et al., 2001. *Manifesto for Agile Software Development.* [Online] Available at: http://agilemanifesto.org [Accessed 03 2011].

Chidamber, S. R. & Kemerer, C. F., 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6).

Elssamadisy, A., 2007. *Agile Measurement - A Missing Practice?* [Online] Available at: http://www.infoq.com/news/2007/07/Agile_Measurement [Accessed 27 February 2012].

Fowler, M., 2004. *UML Distilled: A Brief Guide to the Standard Object Modelling Language.* New York: Pearson Education.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

ISO, 2001. *ISO/IEC 9126-1:2001.* Geneve: ISO.

Kiczales, G. et al., 1997. *Aspect-Oriented Programming.* Jyväskylä: Springer Lecture Notes in Computer Science 1241, Springer-Verlag.

Kitchenham, B. A. et al., 2001. *Preliminary Guidelines for Empirical Research in Software Engineering.* National Research Council of Canada.

Object Management Group, 2010. *UML 2.3 Specification.* Object Management Group.

Olszewski, M., 2012. *Applying Stepwise Feature Introduction to Reengineering of Large Scale Software Systems.* PhD dissertation (to be published) Turku: Turku Centre for Computer Science (TUCS).

Olszewski, M. & Back, R.-J., 2012. *Scrum-Based Agile Development with Stepwise Feature Introduction,* Turku: TUCS Technical Report number 1045.

Petersen, K. & Wohlin, C., 2010. Software Process Improvement through the Lean Measurement (SPI-LEAM) Method. *Journal of Systems and Software,* 8(7).

Reenskaug, T. & Coplien, J. O., 2009. The DCI Architecture: A New Vision of Object-Oriented Programming. *Artima Developer*, 03.

Schwaber, K., 1995. *SCRUM Development Process.* Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA).

Schwaber, K., 2004. *Agile Project Management with Scrum.* Microsoft Press.

Schwaber, K. & Sutherland, J., 2010. *Scrum. The Official Guide.* Scrum.org.