UNIVERSITY OF GOTHENBURG

Designing for Extensibility:

An action research study of maximizing extensibility by means of design principles

Niklas Johansson
Anton Löfgren

Bachelor of Applied Information Technology Thesis

Report No. 2009:053
ISSN: 1651-4769

University of Gothenburg
Department of Applied Information Technology
Gothenburg, Sweden, May 2009

Designing for extensibility:

An action research study of maximising extensibility by means of design principles.

Niklas Johansson
IT University of Gothenburg
Software Engineering and Management
Gothenburg, Sweden
jonikl@ituniv.se

Anton Löfgren
IT University of Gothenburg
Software Engineering and Management
Gothenburg, Sweden
antonl@ituniv.se

Supervisor: Carl Magnus Olsson
IT University of Gothenburg
Software Engineering and Management
Gothenburg, Sweden
carl.olsson@ituniv.se

May 29, 2009

Abstract

This paper presents an action research study on how a set of design principles applicable to object oriented languages can be used to counteract code rot and consequentially enhance system extensibility. The study describes how these principles help support the system quality attributes of modifiability, maintainability and scalability, as well as how these quality attributes correlate to extensibility. Furthermore, it also elaborates on the relationship between extensibility and code rot and how the absence of the first can lead to the latter. The study thus contributes by illustrating how a design science approach can be useful in action research.

Keywords: Action research, design science, extensibility, code rot.

1 Introduction

Our society today is overflowing with data and information about everything and nothing. The Internet has enabled us to do things with a computer which we could only dream of just a couple of years back. The technology we use has also evolved over this time as new concepts emerge and become widely used. To keep up, developers must accept and embrace the fact that systems need to be extended in order to survive. However, not all systems developed with an intent of future extension, extend gracefully. The concept of designing for change has been around since the late 1970s[1]. By this time, many companies were spending big parts of their software budgets on maintenance and that number has been growing ever since[2]. Parnas suggests a set of techniques to design for change which include ”Information Hiding“ and the identification of subsets within the design[1]. Others have since then approached the problem from a more practical point-of-view and have implemented libraries and extensions for programming languages to tackle the problem area of extensible software[3][4]. This study proposes a set of design principles which, when applied,

should have a positive effect on the way software systems react to extension and modification. We also make use of a real life example where the application of our design principles are implemented into a system where extensibility is a primary goal in order to test the validity of our proposal.

The real life example is the development of a system named Metabolism. This system will be implemented by the authors to begin with. Metabolism's main purpose is to map the relations between Free/Libre Open Source Software (FLOSS) development trends and the events that trigger them. For example, what happens to the development within the competing desktop manager projects Gnome[1] and KDE[2], when one releases a new stable version of their system. To do this, the idea is to collect data from the respective projects code repositories and observe around what times the development activity peaks, and based on that, look for events that occurred around that date. The data is then presented through a web-based presentation layer. To involve and capture the users attention the real life events will be gathered by the system, but the users will themselves have influence on what events they find the most likely to have set off any fluctuation in activity. This system is being developed in close collaboration with the Free Software Foundation Europe (FSFE) [3] and the GNU project[4]. We have been working with people from both FSFE and GNU and communication through meetings have been held on a weekly basis. This provides a good environment in which to apply action research. Since we have been working with both GNU and FSFE all the software produced within this project will be released under the GNU GPLv3 license[5] and the system is written completely in Python.

[1] www.gnome.org
[2] www.kde.org
[3] http://www.fsfe.org/
[4] http://www.gnu.org/

2 Problem Area

2.1 Code Rot

A system built from a design that has not taken into appropriate consideration the elements of evolution and extensibility, may well become victim to the phenomenon known as "code rot" or "design rot" and may because of this ultimately be abandoned as the system becomes too hard to maintain and extend. Having to change a system for new features, patching old features, cleaning up obsolete features or even supply a customer with a specified product is all costly work, and it may become even more costly as every change may introduce new problems in the form of bugs and violations of system design.

Martin[6] identifies the following four, non-orthogonal, symptoms of rotting design;

- Rigidity - The software tends to be difficult to change. Changes made requires changes to be made in dependent modules as well.
- Fragility - The software tends to break in many places when it is changed.
- Immobility - The software tends to make it difficult to reuse software parts, both from other projects as well as the same project.
- Viscosity - The software tends to make it harder to employ design preserving methods than methods that do not preserve the design.

The purpose of this study is to identify and assess a number of suitable design principles to minimise the risk for code rot, as a proof-of-concept enabling the obtaining of both practical experiences and theoretical reflection. We are however aware of the fact that software systems can not be kept sane by the influence of design principles alone, and wish to make clear that we are not suggesting this approach as a "Silver Bullet"[7] solution. Other factors also influence the life of a software project, such as its developers dedication to follow the set design or its managers willingness to allow for the design to be followed (i.e. to not stress development in ways that make it hard for developers to adhere to the design without falling behind schedule).

Even so, we believe that the application of and dedication to a reasonable set of design principles may well increase the success probability of a software project in terms of design and code rot avoidance.

2.2 Extensibility

In order to avoid code rot, software needs to be designed with the goal of extensibility in mind. We define extensibility as the ability of a system to be extended with new functionality with minimal or no effects on its internal structure and data flow. The quality attributes presented below are the core contributing factors to extensibility as a system property. Each design principle suggested in this study is related to one or more of those quality attributes in order to achieve the final purpose of extensibility.

To be able to approach the problem of identifying suitable design principles for extensible software, we have started out by identifying the most important quality attributes of such a software system. This study looks in particular at the role of the following three quality attributes as means for the development of design principles useful for acquiring extensibility;

- Modifiability
- Maintainability
- Scalability

Thus identifying that our focus would be to find and apply design principles that supported modifiability[8], maintainability[9] and scalability[10] as well as possible. One could argue that there are more quality attributes that needs to be considered for this study in order to cover the whole spectrum that constitutes extensibility. We agree that there are several other quality attributes that benefit extensibility in some way. We have however so far identified the quality attributes presented here as the primary influences on extensibility, and have for this reason decided to focus on only these three for this study. Furthermore, the time frame we have disables us from testing every possible quality attribute. The sections below will further explain the quality attributes and their definitions. It will also present what design principles map to each quality attribute.

2.3 Modifiability

Modifiability as defined by Bass, Clements and Kazman[8] is determined by how functionality is divided architecturally and by how coding techniques are applied within the code. A system is modifiable when a change involves the least number of changes to the least number of possible elements. We interpret this as striving towards high cohesion[5] and low coupling[6] within the code. We chose modifiability after discussions with our industry contributors and what was envisioned about the systems characteristics. We identified that one of the most important things to think about is that to get support from the community and to get other people contribute to the development one must have code that is easy to modify. If the code is hard to read and understand it is less likely that other developers will take interest in helping the development further. It is also of high importance to be able to conform to new standards and other new inventions. Alfonzo et al. argue that, for these reasons, FLOSS tools must be built so that it is as easy to modify and maintain as possible[11]. Figure 1 shows the relation between the quality attribute modifiability and the set of design principles intended to support modifiability. The design principles shown in figure 1 will be presented in Section 2.6.

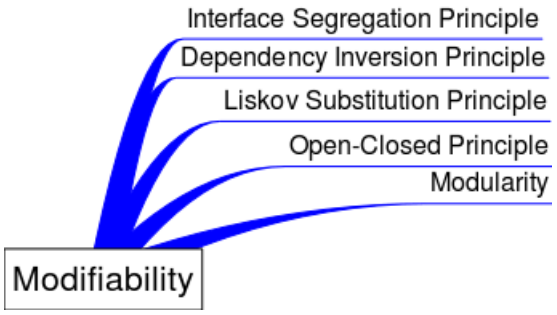


Figure 1: Principles identified to support Modifiability

[5] Cohesion describes how well defined and focused each modules responsibility is.

[6] Coupling describes the level of dependency one program module has to other program modules.

2.4 Maintainability

Maintainability is similar to Modifiability, but we see a distinct difference between the two in their definition. Maintainability as defined by Sommerville[9], is to design a system to allow for the addition of new requirements without risk of adding new errors. We find that these two quality attributes used together complete each other rather than having the same purpose. This also requires the code to have high cohesion and low coupling. We also interpret this as having to take ripple effects into serious consideration, as adding features and correct errors should not raise the risk of adding new errors. Having high maintainability provides a higher probability of a smoother future for the developers and maintainers of a project. In this case specifically since the code is to be released as free software, it needs to be maintainable for the project to have any chance at all of success[11]. Figure 2 presents the relation between the quality attribute maintainability and the set of design principles found to support maintainability.

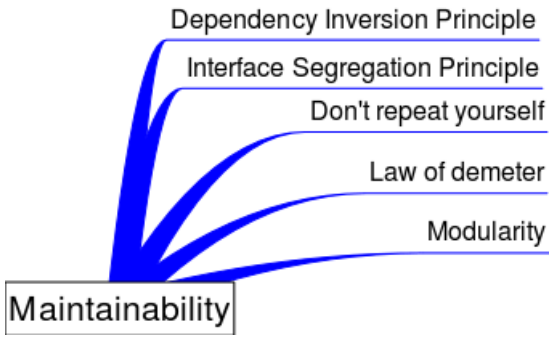


Figure 2: Principles identified to support Maintainability

2.5 Scalability

Scalability as defined by Bondi[10], is "the ability of a system to expand in a chosen dimension without major modifications to its architecture". As our intended system implementation has a user interface that should be presented as a web page, and taking the possible growth of data sources into consideration, this brings forth a set of new problems. For example, what happens if the user load is higher than we had initially expected or that the amount of data that the system needs to harvest is much larger than expected. This presents an obvious need for a scalable system. Scalability is therefore deemed one of the three quality attributes of high importance. Having scalability provides the system with a good possibility to grow if needed. It can be the difference between success and failure. Krasner published an article about typical design problems in Enterprise Resource Planning systems[12] in which he argues that users will not be pleased with having to wait for the system to react to their input, and that these problems can be avoided with proper scalability. When providing a good possibility for scalability, one also opens up for better extensibility by making the system adaptable for many different configurations. Figure 3 presents the relation between the quality attribute scalability and the set of design principles intended to support scalability.

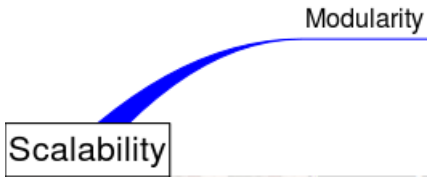


Figure 3: Principles identified to support Scalability

2.6 Design Principles

When we first set out to identify this particular set of design principles we found that Martin[6] had proposed a large set of principles. Some of which are more interesting for use in our work than others. Thus some of the principles Martin presents have been used, but some have been excluded. Furthermore Martin does discuss the Liskov Substitution Principle[13] as well. This study will test Martin's principles with a set of other principles as complement. The reason we chose not to use all of Martin's principles is that we find the set of principles we present more suitable for the purpose of this study. We also find it impossible to take into consideration every principle available because of

the time frame we have available to carry out this study. The set of design principles this study proposes has been chosen primarily for the purpose of providing extensible software and avoid symptoms of code and design rot. We see the chosen set of design principles as a good combination to design for extensibility. Below is a list of the chosen principles;

- P1: Liskov Substitution
- P2: Open-Closed
- P3: Modularity
- P4: Don't Repeat Yourself
- P5: Dependency Inversion
- P6: Interface Segregation
- P7: Law of Demeter

These principles will be further explained in their respective paragraph below.

P1: Liskov Substitution

Liskov Substitution as first presented by Barbara Liskov[13] and refined by Martin[6], is a principle used for easy substitution of one class to a derived class of the same type. So if we want to change class A to an object of class B, class B must be a subtype of class A and provide the necessary methods and signatures. This will for example be applied with the aim to create a plug-in like mechanism for the collection of data from the code repositories of the projects. The idea is that we will create an overall interface with a standardised set of methods which all plug-ins of this kind must implement in order to function with the system. And by doing this it will be easy to add functionality for new version control systems. By using this we are rewarded with both an increase of maintainability and modifiability through the ability to control where changes happen as well as extensibility through the use of a strict design to make it easy to add new classes to the implementation.

P2: Open-Closed

The purpose of Open-Closed is to extend a module without changing the already existing code, as first proposed by Meyer[14] and later, refined by Martin[6]. Martin defines the principle as "A module should be open for extension but closed for modification". The application of this principle will be used to control the stability of the system in a post-release state where working code should not be tampered with, if not absolutely necessary. We argue that the only time you should touch "working" code is if it is not working. Doing so anyway only increases the risk of introducing unnecessary bugs and errors, thus digging a deeper hole for yourself to climb out of. Furthermore its application will strengthen the use of Liskov Substitution by complementing its purpose to increase maintainability and modifiability, thus increasing the extensibility of the software.

P3: Modularity

Booch defines modularity as "[...] the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."[15] In addition, Larman suggests that "(at object level) we achieve modularity by designing each method with a clear, single purpose and by grouping a related set of concerns into a class."[16]

Even though modularity is sometimes considered a quality attribute, we have chosen to include it in our set of design principles for a couple of reasons. First off, we believe it provides developers with enough practical guidance to constitute a principle in this case. Also, after consultancy with our industry collaborator, modularity did not fit into the quality attributes that were requested, which convinced us even further to include it in our design principle set. Furthermore, the use of modularity also provides the system with a means of being scalable, since one module can be easily replaced if needed. Not only scalability gains from the use of modularity, but maintainability and modifiability too. Having a system designed in a way that allows the exchange of modules easily makes it easier to both modify and maintain.

P4: Don't Repeat Yourself (DRY)

Hunt and Thomas[17] defines the DRY principle as follows;

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

The thought is that by adhering to this principle, the system being developed will be both easier to maintain and to understand. By not having the same information expressed in multiple places, developers do not have to keep track of all the places where that information is expressed. As a consequence, developers need not worry about breaking the system by changing the information. The principle applies to all aspects of a software project, including specifications and documentation, processes and programs. Applying this principle limits the places where you have to change code to correct errors and lowers the risk of introducing new bugs and causing a ripple effect. Thus increasing modifiability and maintainability through the application of a strict code structure. Furthermore it enhances the possibility to reuse the code.

P5: Dependency Inversion

Dependency Inversion is a principle proposed by Martin[6], which implies that one should depend upon abstractions rather than upon concretions. The strategy is to depend upon interfaces rather than the actual implementation of the interface. Which provides the possibility to easily exchange, for instance, faulty implementation to correct implementation without having to change the code that depends upon the interface. This further complements the use of both Liskov Substitution and Open-Closed. It also provides a good structural basis for the treating of inheritance and abstract classes. Being able to exchange modules, classes and other code entities brings forth a better growing ground for maintainability and modifiability.

P6: Interface Segregation

Martin also proposes Interface Segregation[6], which is used to further structure the use of interfaces and abstract classes. Martin suggests that when more than one client class accesses the same service (that contains client specific methods and functions) each client should be assigned its own interface to this service. The application of this principle will render the code easier to change and maintain, thus supporting both maintainability and modifiability. It also creates a more readable code structure which is always good when you expect other people to contribute to the code base. Furthermore it allows further support to the application of Law of Demeter (see below).

P7: Law of Demeter

The Law of Demeter as propounded by Lieberherr, Ilolland and Riel in 1988[18] implies that software entities should only speak directly to their closest neighbour. This can be simplified into saying for example, that when you drive a car you accelerate by pressing the gas pedal and not by telling your motor directly to throttle. Another example would be that you use the steering wheel to turn your car, and not by telling each individual wheel directly to turn a certain amount. Applying the Law of Demeter, the design will be tightly connected to the flow of data, making it easier to identify which parts of the system that should be responsible for what tasks. This also makes it easier to identify flaws in the design while implementing. This will of course lead to redesign, but we believe that this is a better option than to be stuck with a system that is faulty by design.

Principle Map

Figure 4 describes how the design principles are used to support the quality attributes and how the quality attributes ultimately is intended to support the idea of extensibility. We can see that some of the design principles are used to support both modifiability and maintainability. This is a result of the likeness between modifiability and maintainability, where some of the design principles gives advantages to both purposes. Moreover the need for high cohesion and low coupling are strong in both these quality attributes and therefore principles that enable these two practical advantages all are considered to support these quality attributes. The sorting of the design principles does not imply any difference in importance, but is only a means of making the map more readable.

To understand the importance of each branch in the map one should read from left to right. This is how we

have thought when developing this set of design principles. After discussions with our industry collaborator we came to the conclusion that we wanted a system that was extensible and by this we identified the quality attributes that help us accomplish this. From these quality attributes we looked at possible design principles to support each of them. As mentioned above it has been identified that some of the principles support only one quality attribute, whilst other support two or even all of the quality attributes.

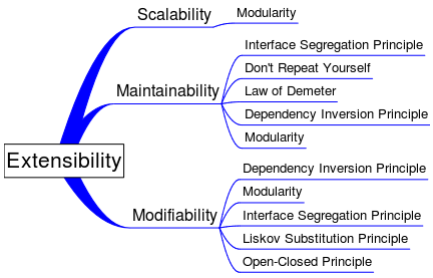


Figure 4: Design Principle Mapping

3 Research method

3.1 Action Research

The research was carried out in an iterative manner, in order to allow us to refine our set of design principles to further investigate the correctness of our choices. In short, our research model resembles that of action research as defined by Susman [19]. This process is also illustrated in figure 5. Our use of action research has been adapted in order to fit the rather narrow time frame given to this research project. The practical impact this deviation has had is that a smaller number of iterations has been used rather than what is usual in action research. All cycles of the research process has thus not be considered in our research. The advantages gained from using the action research model has however outweighed the fact that this project did not match the template perfectly. O'Brien states that;

> "Participants in an action research project are co-researchers. The principle of collaborative resource presupposes that each person's ideas are equally significant as potential resources for creating interpretive categories of analysis, negotiated among the participants. It strives to avoid the skewing of credibility stemming from the prior status of an ideaholder. It especially makes possible the insights gleaned from noting the contradictions both between many viewpoints and within a single viewpoint."[20]

This suited our research project well since a large amount of industry collaboration comes naturally to the research we have performed. Furthermore, Baskerville suggests that "Action research is empirical, though the collected data is typically qualitative and interpretive"[21], thus also supporting the way in which we have collected and analysed our data as is presented in section 4.

Additionally, we will present examples of how our design principles have influenced the actual design of the system at a practical level in order to further illustrate their impact.

Translated into terms of our research project, the phases present in the research model shown in figure 5 are what we worked from. These activities are further described in their respective subsection in this chapter.

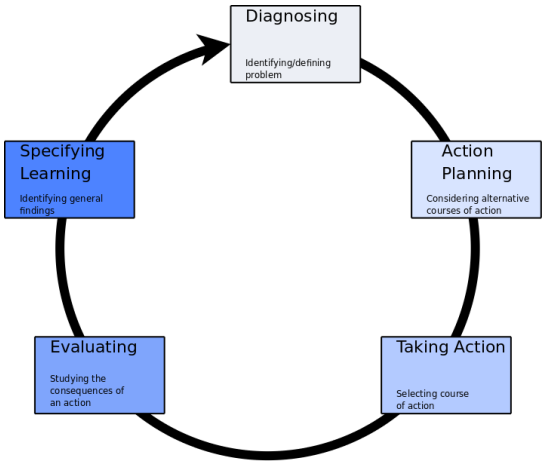


Figure 5: Action research model (adapted from [19])

3.2 Diagnosing

During the diagnosing phase we had already established good interaction with our industry collaborators. We had set up weekly meetings for communicating the work that had been done and what we were currently doing as well as planning for the next step. These meetings also served as a kind of acceptance meeting for any changes that had come up as our work progressed. Furthermore, we identified modifiability, maintainability and scalability as good quality attributes to use when designing for extensibility.

3.3 Action Planning

This phase was used primarily to identify which design principles we saw fit for the purpose of providing extensibility and thus counteract code rot. We spent a large amount of time studying the related research and trying to come to terms with where the design principles would fit in, and to what extent they would be useful. Moreover, we looked at possible frameworks to help us achieve the purpose of our application.

3.4 Taking Action

The taking action phase was used to create and refine the system design, the architecture and to implement the application. We based our design and implementation upon the design principles we identified in the action planning phase. The architecture can be seen in figure 3.4. When the design and architecture was complete and perceived as embodying the design principles identified in the action planning phase, we proceeded to implement the system.

3.5 Evaluating

The evaluating phase was used for interviews. We invited a group of experts to assess our design and the specific implementation of the principles. We received overall positive results (described in better detail in section 4.1). Furthermore, we reflected over how well we could see our design principles working ourselves. As an example of this, we can highlight the bug fixing process. Mostly we only had to change code in one module to fix bugs. Thus, showing that ripple effects had been prevented in a good manner. We also reflected upon our choice of programming language for the implementation. We had chosen Python, as we had some prior knowledge of the language and had found that Python oftentimes gives a rapid development process. The time it took us to implement big parts of the system was also surprisingly short to our customer. This allowed us to spend more time on refactoring and refining the code base[7].

3.6 Specifying Learning

In this phase we analysed the results from the interviews conducted during the evaluating phase. We also reflected upon what we had learned throughout of all of the prior phases. We looked at what could be improved for the next cycle in terms of design principles. Furthermore, Walsham[22] states that there are four different types of information system research generalisations.

- Development of concepts
- Generation of theory
- Drawing of specific implications
- Contribution of rich insight

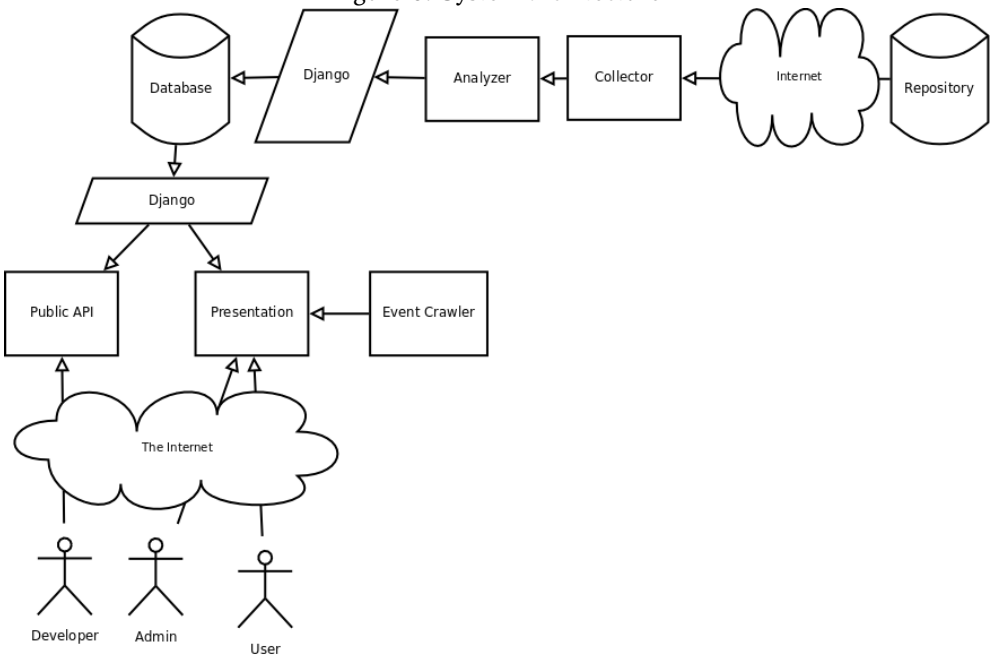
We found ourselves in between two of these fields, in particular Drawing of specific implications and Contribution of rich insight. We do not contribute with rich insight directly, but most of our work is based on prior rich insight contributions from other authors. This is specifically true for the various design principles presented in this study. We do on the other hand contribute directly with the relationship between our design principles and the system property of extensibility.

4 Analysis and Discussion

To evaluate whether our principles have had a positive or negative impact on the system, we have conducted interviews with potential extenders of the system. We

[7] Available at: http://itupw056.itu.chalmers.se/metabolism_docs/

Figure 6: System architecture



analysed their answers in order to find flaws in any assumptions we have made about what constitutes good extensible software systems as well as any shortcomings in our design principle set. We have also carried out experiments to test the extensibility of the system in order to directly test its extension capabilities. These experiments have been on a software testing level. We have investigated the use of other modules and packages, and their effect on our system's performance. This may also come into use when proving the modularity of the system. We have made use of test-driven development (TDD) as described by Beck[23] in order to easily detect system breakage when performing extensibility experiments. Test results have thus given us additional indications as to whether our system is easy to extend or not.

For the purpose of comprehensiveness of this study, we have chosen to analyse only a subset of the modules of the system. A diagram of this subset is shown in figure 7 in Appendix A. This particular subset has been chosen because these modules form the foundation of the system and are as such the most likely target for significant extension. We see an extension as significant when it provides the system with new functionality that changes the scope or alters the requirements of the system. This module controls the interaction with project source code repositories, and it is responsible for the gathering of project data.

To reiterate, we have used a mixed approach to data collection in our research project. Both qualitative and quantitative data have been collected and analysed in order to get a more holistic view of our results. Our data collection and data analysis relates to the use of action research in the sense that many data sources was used. Since we were only performing one cycle of the action research method in the scope of this project we can make use of the information gathered from the data when developing the system further.

In order to minimise violations of the DRY principle, we made use of a tool, Clone Digger[8], that detects and reports on duplicated and cloned code segments, e.g, a certain method being called with the same arguments

[8] http://clonedigger.sourceforge.net

from several different, unrelated points in the program. The tool achieves this by using various algorithms presented in [24]. We have analysed the output from this tool regularly in order to find such violations at an early stage, enabling us to refactor duplicate segments into cleaner methods and classes and achieving a better adherence to the DRY principle. Clone Digger grades the clones by a factor where one is segments that are identical to each other. These identical segments have been actively located and refactored throughout the development to adhere to the DRY principle. This is has proven very powerful since each person contributing to a project can not have a perfect understanding of what other developers are producing.

During the development, it became evident that the principle of Dependency Inversion was not applicable for the specific implementation we present here. In our case, third-party tools largely handled the parts where this principle normally would have been applied. We nevertheless feel that this principle is relevant for consideration, but it highlights an important factor still - that design principles may not all be directly applicable for implementation, as long as they are seriously considered.

4.1 Interviews

In order to validate that all applicable design principles had been applied in an adequate manner, we asked three different industry experts to evaluate the aforementioned module subset. As we have already discussed, the principle of Dependency Inversion is not part of this evaluation. Nor did we ask the industry experts to evaluate the application of the Open-Closed principle as it is not a suitable target for static evaluation. Since our interpretation of the Open-Closed principle is that it should be applied once the code has been functionality tested and released it is only used post-release. The system built during this study has not yet reached a state where it can be considered ready for release and we see Open-closed as a principle to be applied in the future. For each of the applicable design principles, we asked the experts to rate their agreement level with the statement; "The design principle is applied well to the software module" on a scale from one to five (a rating of one meaning "strongly disagree" and a rating of five meaning "strongly agree"). The results of this analysis are shown in table 1. The table shows that the experts generally consider the design principles to be well implemented, even though there are a few deviations. Expert 1 considers all of the design principles except for DRY to be well implemented. The expert however comments that Liskov Substitution may be somewhat hard to evaluate due to ”the small amount of inheritance on the same "level"“. Expert 1 also suggests that there are hidden duplication present in the module subset in the form of different functions performing the same tasks, thus violating the DRY principle.

Expert 2, on the other hand, did not find any violations of the DRY principle. The expert however comments that he had trouble finding any examples of the implementation of Interface Segregation. The expert also mentions that the reason for his grading of four on Modularity stems from a number of static attributes in one of the classes. He infers that this leads to a higher level of coupling than necessary. Additionally, this expert has graded the Liskov Substitution principle lower than any of the other experts in this study but has not provided any further explanation as to why this is. Expert 2 furthermore proposes the future consideration of the Dependency Injection[25] principle, a principle which is quite similar to Dependency Inversion, as they are both inversion of control principles.

In the light of these results, the answers given by Expert 3 may be somewhat surprising. It is hard to say how this deviation in grading came to be, since no comments on the specific design principles were provided. Expert 3 did on the other hand provide general comments. He states that from his own experience the use of these design principles or any design principles for that matter, should be considered separately for each project you find yourself working on. Thus basing the decisions on what the future holds for each system individually. This is a good point indeed, and we fully agree that there are no perfect set of design principles to use for each project or product.

Two of the experts also commented on the use of design principles in general and offered some insight on the subject. Similarly to what was suggested in section 2.1, they believe that the sanity and success of a software system is not solely dependent on design princi-

Table 1: Inquiry results

Subject	Liskov Substitution	Modularity	DRY	Interface Segregation	Law of Demeter
Expert 1	5	5	3	5	5
Expert 2	3	4	5	3	5
Expert 3	5	5	5	5	5

ples but rather on the individual assessment of what is needed for each project and product. There was also comments discussing the pace at which the field of information systems development moves, and that this would present a problem for the application of the kind of solutions presented in this article. We do not necessarily agree to this statement, since this proposition is not in any way language or system specific. Our implementation has been done exclusively in the Python programming language, but the theories should be applicable to any object oriented language.

5 Conclusion

We find that our implementation of the proposed design principles is to be considered successful. The results of the expert inquiry shows that there are different opinions to the level of how well each principle were implemented. But since all principles that were investigated by the experts have an overall result that is positive we interpret the results of the inquiry as positive. It has also become apparent that each project and product has its own set of needs, thus rendering each project in some way unique. This has to be considered when making design decisions but the set of design principles we propose in this article is a good starting point when striving towards extensibility. As has become evident from the implementation presented in this paper, not all of the suggested design principles fit the scope of all projects. However, as long as all principles are seriously considered, this set carry several advantages by reducing the risk for code rot when developing a system that may require future extension. The research upon which this study has been based, in particular the design principles have already proven the advantages gained when the specific principles are applied. We are therefore confident that this set of principles carry a good ground for extensibility.

The natural step for the next cycle of this research is to look at the design principles that did not apply to this study and study their appropriateness further, as well as extend the application. Extension for the application would include, but are not limited to, support for other repository systems and an open API for external resources. The reason for an open API is that it would be interesting to see if the design principles would have an effect on how well an API evolves over time. The system that was produced during this study is still in early development, but support for Subversion repositories has been added and is functional. Some screenshots of the alpha version can be found in Appendix B.

References

[1] D. Parnas, “Designing software for ease of extension and contraction,” *IEEE transactions on software engineering*, pp. 128–138, 1979.

[2] J. Koskinen, “Software Maintenance Costs,” 2003. Acessed 2009-04-06 at: http://www.cs.jyu.fi/ koskinen/smcosts.htm.

[3] L. Tokuda and D. Batory, “Automated software evolution via design pattern transformations,” in *Proceedings of the 3rd International Symposium on Applied Corporate Computing*, 1995.

[4] M. Zenger, “Evolving software with extensible modules,” in *International Workshop on Unanticipated Software Evolution*, 2002.

[5] F. S. Foundation, “GNU General Public License,” 2007. Accessed 2009-04-06 at: http://www.gnu.org/licenses/gpl.txt.

[6] R. Martin, "Design principles and design patterns," *Object Mentor*, 2000.

[7] F. Brooks, "No silver bullet: Essence and accidents of software engineering," *IEEE computer*, vol. 20, no. 4, pp. 10–19, 1987.

[8] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

[9] I. Sommerville, *Software Engineering 8*. Addison-Wesley Professional, 2007.

[10] A. Bondi, "Characteristics of scalability and their impact on performance," in *Proceedings of the 2nd international workshop on Software and performance*, pp. 195–203, ACM New York, NY, USA, 2000.

[11] O. Alfonzo, K. Domínguez, L. Rivas, M. Pérez, L. Mendoza, and M. Ortega, "Quality Measurement Model for Analysis and Design Tools based on FLOSS 19th Australian Software Engineering Conference (ASWEC 2008)," in *Proceedings of the 19th Australian Software Engineering Conference (ASWEC 2008)*, vol. 1, pp. 258–267.

[12] H. Krasner and K. Consulting, "Ensuring e-business success by learning from ERP failures," *IT Professional*, vol. 2, no. 1, pp. 22–27, 2000.

[13] B. Liskov, "Data abstraction and hierarchy," 1987.

[14] M. Bertrand, *Object oriented software construction*. Prentice Hall, 1988.

[15] G. Booch, *Object-oriented analysis and design*. Addison-Wesley Reading, MA, 1996.

[16] C. Larman, *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2004.

[17] A. H. et al., *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

[18] K. Lieberherr, I. IIolland, and A. Riel, "Object-oriented programming: An objective sense of style," 1988.

[19] G. Susman, "Action research: a sociotechnical systems perspective," *Beyond method: Strategies for social research*, pp. 95–113, 1983.

[20] R. O'Brien, "An overview of the methodological approach of action research," *Unpublished paper to Professor Joan Cherry, Course LIS3005Y, Faculty of Information Studies, University of Toronto. April*, vol. 17, 1998.

[21] R. Baskerville, "Investigating information systems with action research," *Communications of the AIS*, vol. 2, no. 3es, 1999.

[22] G. Walsham, "Interpretive case studies in IS research: nature and method," *European Journal of information systems*, vol. 4, pp. 74–74, 1995.

[23] K. Beck, *Test-driven development: By example*. Addison-Wesley Professional, 2003.

[24] P. Bulychev and M. Minea, "Duplicate code detection using anti-unification," in *Spring Young Researchers Colloquium on Software Engineering, SYRCoSE*, vol. 2008, p. 4, 2008.

[25] M. Fowler, "Inversion of control containers and the dependency injection pattern," *Actualizado el*, vol. 23.

Appendix A

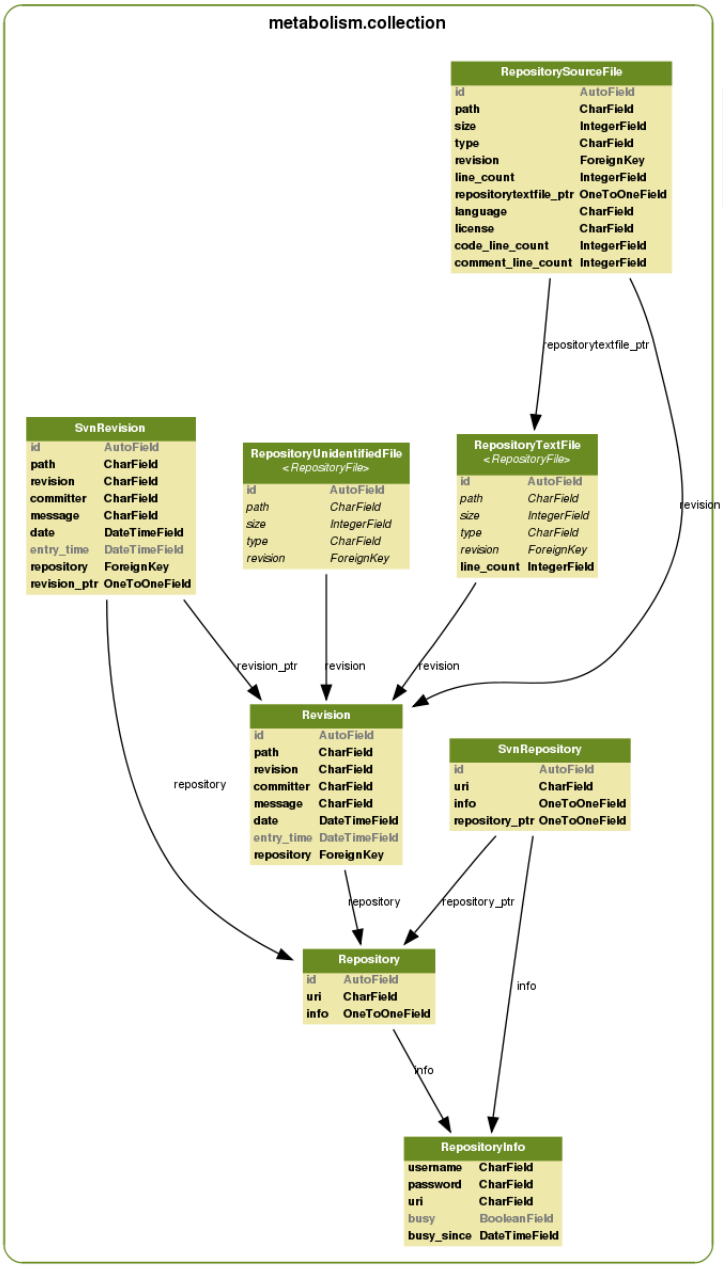


Figure 7: Module subset used during the inquiry

Appendix B

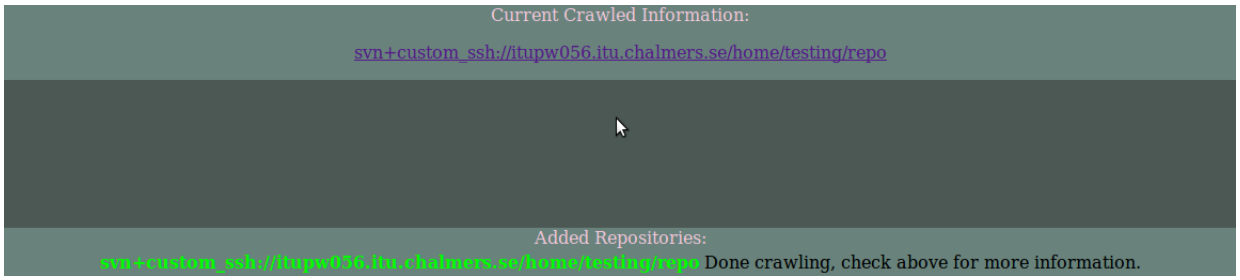


Figure 8: Metabolism presentation layer

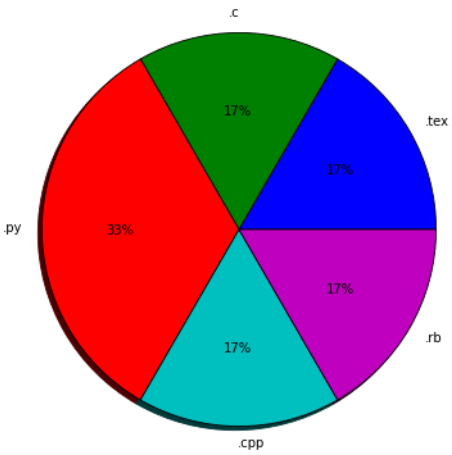


Figure 9: Repository Language Distribution Graph

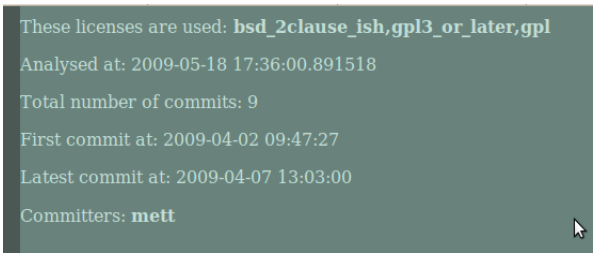


Figure 10: Repository Information Summary