A Principle Language for Object-Oriented Design

Master's Thesis

Christian Rehn

AG Softwaretechnik
Prof. Dr. Arnd Poetzsch-Heffter
Dipl.-Inf. Patrick Michel

April 15, 2013

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie die Zitate deutlich kenntlich gemacht zu haben.

Kaiserslautern, den 15. April 2013

Abstract

Software design is a non-trivial task. Especially for making sound design decisions there is still little guidance and communicating the reasons behind those decisions also bears difficulties. There are many software design principles which help here but in the past they have mainly been discussed and examined in isolation. The goal of this thesis is to interconnect these principles in order to support design decision making. Just like patterns are interconnected to pattern languages, principles can be interconnected to principle languages where the consideration of one principle inevitably leads to the consideration of other principles. For this thesis such a principle language has been constructed and documented in a wiki, which will go online shortly. A snapshot of the wiki is included in this thesis. Along the principle language an analytic design approach has been developed which uses the principle language to guide developers making design decisions. In order to evaluate the approach and the language, two experiments have been conducted which show promising results. The first experiment examines the utility of the approach compared to Fowler's code smells and Martin's SOLID principles. And the second experiment indicates that a consequent usage of the principle language may have a positive effect on software quality. This thesis opens up a new area of research. There are still many unanswered questions. But this work lays the foundation for answering them.

Zusammenfassung

Der Softwareentwurf ist eine nicht-triviale Aufgabe. Insbesondere für das Treffen von Entwurfsentscheidungen besteht immer noch wenig Unterstützung. Auch das Kommunizieren der Gründe für solche Entwufsentscheidungen ist nicht einfach. Es gibt viele Entwurfsprinzipien, die hier helfen, jedoch wurden diese bisher hauptsächlich einzeln und unabhängig voneinander untersucht. Das Ziel dieser Arbeit ist es, solche Prinzipien miteinander zu vernetzen, um so bei Entwurfsentscheidungen zu helfen. So wie man Muster zu Mustersprachen verbindet, kann man auch Prinzipien zu Prinzipiensprachen verbinden, sodass das Betrachten eines Prinzips unausweichlich zu weiteren Prinzipien führt. Für diese Arbeit wurde so eine Prinzipiensprache konstruiert und in einem Wiki dokumentiert, das in Kürze online gehen wird. Ein Ausschnitt aus dem Wiki ist in dieser Arbeit enthalten. Neben der Prinzipiensprache wurde ein analytischer Entwurfsansatz entwickelt, der die Prinzipiensprache nutzt, um Entwicklern beim Treffen von Entwurfsentscheidungen zu helfen. Um sowohl den Ansatz, als auch die Sprache zu evaluieren, wurden zwei Experimente durchgeführt, die vielversprechende Ergebnisse zeigen. Das eine Experiment untersucht die Nützlichkeit der Prinzipiensprache im Vergleich zu Fowlers code smells und Martins SOLID-Prinzipien. Das zweite Experiment deutet an, dass die konsequente Nutzung der Prinzipiensprache positive Effekte auf die Softwarequalität haben könnte. Diese Arbeit eröffnet ein neues Forschungsfeld. Es gibt noch viele unbeantwortete Fragen, aber diese Arbeit legt die Grundlage zu deren Beantwortung.

Contents

1 Introduction

1.1 Motivation

Software design is an inherently complex task. It requires knowledge, skill, and experience. First of all the domain, the technology, as well as the programming paradigm and design approach (e. g. object-oriented design) have to be known and well understood. Though not trivial, such knowledge is relatively easily obtained as knowledge is teachable. On the other hand skill and experience, which are needed to make good design decisions, are not directly teachable. So it takes a long time to gain good design competences. The same problem arises when communicating about design. Furthermore as experience is not tangible, it is difficult to communicate. So discussing which design solution is better in a given context can be difficult. "Experience" alone is not a convincing reason to prefer one solution over another one. So it's not only difficult to do design and to make design decisions but also to explain the reasons for these decisions and to talk about them in a development team.

As a resort often scenarios are used. If one solution to a design problem is better than another one, there has to be some situation where this advantage takes effect. So scenarios are constructed that show the advantage. Such a scenario could for example be a speculative future enhancement or change of a specific part of the software. If one solution makes this easier than another one, it is better in this case.

But this scenario-based approach has several disadvantages. A scenario is just one concrete situation. There may be others where the solution has a contrary effect. So in order to get a complete picture, many different scenarios have to be considered. Furthermore constructing suitable scenarios can be a time-consuming task. So constructing scenarios for every design decision is not feasible. A more light-weight approach is needed for making the every day design decisions.

So another approach to do software design is to base it upon certain principles. A set of principles may be used to describe a given design problem, assess possible solutions and make the corresponding design decision. Such principles like for example →4.5.2 *Low Coupling* (LC), →4.4.2 *High Cohesion* (HC) or →4.5.1 *Tell, don't Ask/Information Expert* (TdA/IE) give advices on how to design software. Using these principles one can reason about software design in a more abstract manner compared to using concrete scenarios. This does not make scenarios obsolete. Especially for functional requirements as well as for certain non-functional requirements scenarios are still necessary. But principles can serve as a general way of thinking for the every day design decisions. Since the early days of software engineering seasoned designers and well-known researchers have condensed their experience into easily memorable principles and rules. Using principles to share knowledge has a long tradition in the history of software engineering. Some of these principles are well known and

have been subject to scientific examination. Others still remain widely unnoticed.

But while principles in general are widely used to address particular design aspects, it is still not completely understood how to use them in general. Furthermore principles mainly have been described in isolation. But since usually several principles can be considered for a given design problem, it is worthwhile examining how principles interrelate and interact with each other as well as how a useful subset of generally applicable principles can look like.

Research on principles can help making principles a basis for teaching software design, reusing design experience and communicating and documenting design decisions. By learning how to use software design principles, one could gain design competences easier and faster by reusing experience made by others. Such a reuse of design experience would certainly have similar benefits as other forms of reuse, namely increased quality and simultaneously reduced cost and time. Also already experienced designers could benefit from the use of principles when they form a common vocabulary which enables them to talk about design decisions in a clear way. So principles help communicating and documenting design decisions. But at first research is needed to form such a common set of terms.

The goal of this thesis is to examine principles and their relationships, to describe an approach for using them as a guideline for object-oriented design and as a means to communicate about design decisions.

1.2 Terms and Definitions

Some of the terms used here to describe principles originate from research on patterns. The term *pattern* was coined by the architect Christopher Alexander during the 1970ies to describe common solutions for building towns and houses [1]. Ten years later Kent Beck and Ward Cunningham transfered the idea to software development [2]. The idea gained popularity in the following years most notably through the books "Design Patterns: Elements of Reusable Object-Oriented Software" [3] and "Pattern-Oriented Software Architecture: A System of Patterns" [4].

In the latter book *pattern* is defined as follows: "A *pattern [...]* describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate." [4, p. 8]

There are several other popular definitions, that slightly differ, but all are about proven, reusable solutions to common problems in certain design contexts. Although rarely stated explicitly, there is a difference between a *pattern description* and the *pattern* itself. In the following it will be helpful to distinguish these two terms:

Definition 1. *A **pattern** is an often-used, proven solution to a recurring problem in a certain context.*

Definition 2. *A **pattern description** is a document that describes a pattern by giving at least the following information:*

- *a name for the pattern,*
- *the context in which the pattern applies,*
- *the problem the pattern solves,*
- *and the solution.*

Furthermore a pattern description typically also gives a motivation, some example code, possible variations of the pattern, known uses, etc.

This distinction makes it clear that patterns are there—whether they are described or not. They are simply recurring solutions, that have to be described in order to communicate them and make them reusable. This also means that patterns normally are not constructed but discovered. Someone realizes that a recurring problem has been solved basically in the same way over and over again. And after the pattern has been found, it can be documented using a pattern description. For convenience a pattern description is often called "pattern" too. But actually it is more than that.

"Observer" for example is a design pattern. It solves the problem how to inform an unknown set of dependent objects when a dependee object changes state. A description of the pattern can be found in [3].

In order to support finding suitable patterns for a given problem, pattern descriptions are collected in pattern catalogs.

Definition 3. *A **pattern catalog** is a collection of several related pattern descriptions, where each description has the same structure.*

Typically pattern catalogs describe sets of patterns of a certain domain, a level of abstraction, or otherwise having a certain commonality.

The original idea by Christopher Alexander was to construct not only a pattern catalog but something he calls a pattern language [1]. A pattern language in Alexander's sense interconnects the patterns in a way that forms a step-by-step guide for a designer. The patterns form a decomposition structure that comprises all relevant problems that occur during design. It precisely determines which design decisions to take in which order. Alexander claims to have constructed a complete pattern language for architecture. So using his pattern language a layperson should be able to make all design decisions necessary to design a room, a house and even towns and regions [1].

It is doubtful whether such a pattern language is possible for general-purpose software design. However it is useful to interconnect several pattern descriptions. For solving a concrete design problem, not only one pattern might be considered but several alternatives. Furthermore patterns are often not applied in isolation but combinations of several patterns are used to solve more complex problems. Therefore it is helpful, when a pattern description refers to possible alternatives as well as to complementary patterns the pattern may be combined with. By doing so, a network of patterns is created that forms a more realistic kind of pattern language:

Definition 4. *A **pattern language** is a pattern catalog where each pattern is linked to those other patterns it is related to, such that the consideration of one pattern automatically leads to alternatives and complements.*

Because these pattern languages differ from those described by Alexander, they are sometimes also called **pattern systems** [4].

Principles as discussed in this thesis are a more general and universal way to communicate experience. Nevertheless they can be described in a similar way:

Definition 5. *A **principle** is an informal rule that tells whether one solution is better than another one with respect to a certain aspect.*

Definition 6. *A **principle description** is a document that describes a principle by giving at least the following information:*

- *a name for the principle,*
- *a description of the rule,*
- *and a rationale that describes why the rule holds.*

Furthermore a principle description should also contain examples, usage hints, etc.

The KISS principle ("keep it simple stupid") for example says that a simpler solution is better than a complex solution as it is easier to write, to read and thus to change. A description of this principle can be found in section 4.3.2.

While patterns and pattern languages have been subject to thorough research, basic principles of software design exist much longer but have mainly been discussed in isolation. Only little work has been done to examine how principles interrelate. One could also imagine principle catalogs and principle languages similar to pattern catalogs and pattern languages. Exploring this idea is the subject of this thesis.

Definition 7. *A **principle catalog** is a collection of several related principle descriptions, where each description has the same structure.*

Definition 8. *A **principle language** is a principle catalog where each principle is linked to those other principles it is related to, such that the consideration of one principle automatically leads to others which are likely to be relevant in the same context.*

In order to find a set of principles which is applicable to a certain design problem, one can start with one relevant principle and the principle language helps to find other related principles which fit into the demanded set. Complementary and contrary principles help finding further aspects to consider whereas generalizations and specializations may be used instead of a previously considered principle in order to find a level of abstraction that fits the problem better.

Section 2.3.1 describes further aspects of the relation between patterns and principles.

Several principles deal with the decomposition and interaction of classes, methods, procedures, functions, etc. In order to abstract from the concrete syntactic element—be it a class, a method, a procedure, a function, an executable or the like—the term "module" is used here:

Definition 9. *A **module** is a piece of code that carries a name and is syntactically distinguished from other parts of the code.*

Another important notion is "interface". In particular there are two terms, both named "interface". First of all there is the concept of an interface as an interaction point and second there is the language feature `interface`:

Definition 10. *An **interface** is an interaction point between modules. It's a generic concept that describes everything which is used in order to use a module.*

Method signatures, for example are part of the interface of a class.

Definition 11. *An `interface` is a language construct of several object-oriented programming languages. It is resembles classes which only have abstract methods.*

For clarity reasons the two terms are distinguished in this thesis by using different fonts. The concept interface is always written like the rest of the text. But whenever the language construct is meant, a monospace font is used: `interface`.

The goal of this thesis is to create a principle language for object-oriented design. This defines the level of abstraction. The term is defined here as follows:

Definition 12. ***Object-Oriented Design** (OOD) is the task of defining how the software works on a class level which comprises the decomposition into classes, defining their interfaces, their method signatures, as well as their internal structure.*

The principle language to be constructed shall guide design decisions.

Definition 13. *A **design decision** is any decision a developer has to make on the level of software design.*

In agile contexts OOD is seen as a part of coding while plan-driven development rather sees it as a separate development phase before the actual coding. Nevertheless roughly the same decisions have to be made.

This definition of OOD also means that everything that is confined to a single method is not a part of OOD anymore and thus out of scope for this thesis. Decisions on algorithms, on language constructs to use, on how to implement a certain algorithm, etc. are not relevant here except if such a decision has an influence on other modules, too. These decisions are too low-level. They may be guided by other principles but this is not a part of this thesis.

Moreover there are also decisions which are too high-level to be considered here. These include decisions on requirements and architecture. Requirements specification tasks are not technical enough to be part of OOD. And although architecture is normally considered to be part of design, it is also not considered here. The decisions on an architecture scale

normally concern subsystems, layers, packages and other groups of classes. Architecture is normally not concerned with single classes and methods. So only these lower-level design decisions are treated here.

Object-oriented design also means that the principle language only applies in an object-oriented context. Typically an object-oriented programming language, like Java or C++, is used, but the important point is that an object-oriented way of thinking is applied. Procedural programming, functional programming and other programming paradigms are not considered in this thesis.

Many of the principles discussed here are about maintainability of software. A serious threat to maintainability are the so-called ripple effects:

Definition 14. *A **ripple effect** occurs when one change in one module makes further changes in other modules necessary which in tun may also produce further changes.*

So a change "ripples" through the code of the software. This is generally undesirable because these additional changes increase the effort spent on maintenance and are also error-prone because some of these changes may be forgotten or may be made incorrectly.

1.3 Basic Idea

The basic idea of using principles as described and examined in this thesis is to use them for making design decisions and to judge whether one solution to a problem is better than another one in a certain context. The claim is that it is possible to describe all relevant aspects of a given design problem using a characterizing set of principles.

The principles tell which aspects to consider while making the decision. One principle might focus on the advantages of simplicity, another one might be about generality and a third one considers effects of inheritance. These principles are usually conflicting which results in the need for a trade-off. For example a solution typically cannot be likewise simple and generic. If it is generic, it is not simple anymore and vice versa. So the designer has to find a suitable compromise. Experienced designers do so intuitively but communicating this intuition can be difficult. Principles help to explain why a certain solution is better than another one.

In order to help inexperienced designers to think about all relevant aspects of a design problem and in order to give experienced designers a common set of vocabulary to explain design decisions, a principle language can be used. Such a principle language wires together several principles so that the consideration of one principle leads to others which might be relevant in the same situation. By following the relationships in the principle language, a characterizing set of principles is obtained which describes the advantages and disadvantages of the possible solutions. The designer can then use this information to make a sound decision.

In this thesis this approach is explained in more detail (chapter 2), a principle language is constructed (chapters 3 and 4) and the result is discussed (chapter 5). Two experiments are conducted in order to evaluate approach and language (chapter 6) and in the end an outlook describing further research possibilities is given (chapter 7).

2 Approach

2.1 Supporting Design Decisions

2.1.1 Generative vs. Analytic Design Approaches

Software development involves frequent design decisions. The designer has to decompose the system to into subsystems and modules. Furthermore suitable interaction mechanisms between the modules have to be defined, appropriate data structures have to be found and so on. Each of these tasks or "problems" comprises a decision on how to solve it. Normally there is more than one possible way to solve a given problem (several possible decompositions, interaction mechanisms, data structures, etc.), so a decision has to be made. The designer has to constantly assess possible solutions and to judge which one fits best to the given context.

There are several approaches that help the designer to fulfill this task. These can be grouped into two kinds: generative design methods and analytic decision techniques. First of all the designer has to generate a solution matching the problem to be solved. Given a certain problem description, generative design methods tell, how to construct a solution step by step. As there is no simple step-by-step method that automatically creates perfect results and there is not much hope that such an approach will show up suddenly [5], the designer now has to judge whether the generated solution is good enough or whether he or she needs to look for an alternative. The aforementioned analytic techniques help judging this. Furthermore if eventually there are several possible solutions, they have to be compared so the solution that fits best can be chosen. This is a second application of analytic design techniques.

The approach discussed in this thesis is an analytic design technique. It does not generate a solution but it helps comparing and judging them.

2.1.2 Judging Design Solutions

There are design decisions on different scales. Some are trivial (like deciding upon the method identifiers or maybe parameter lists). Others are non-trivial but need to be made on a daily basis nevertheless (deciding which data structures to use, how to decompose modules, etc.). And lastly there are also large-scale architectural design decisions (e. g. deciding upon the architectural style, the number of tiers, the middleware to use, etc.), which have a huge impact on the quality of the system.

While making trivial design decisions, like naming methods, may also require some skill, it is mostly not worthwhile to invest too much in it. These trivial decisions only have little

impact and are also easily revocable. On the other hand it may have a larger impact when a large amount of these trivial design decisions is done wrong. This is a case where skill is simply irreplaceable.

For large-scale architectural decisions the situation is completely different. These decisions typically have a massive impact on the quality of the system, so investing large efforts may pay off here. There are several approaches dealing with these cases. They are typically scenario-based like the *architecture tradeoff analysis method* (ATAM) [6]. At its heart these approaches are about systematically examining the architecture using scenarios that show how the system would behave in certain cases.

Between the trivial design decisions and the large-scale architectural design decisions there are many non-trivial design decisions, which have a considerable impact but still have to be carried out on a daily basis. Scenario-driven approaches like ATAM impose a huge effort which do not pay off in these cases even when steps like stakeholder meetings are left out. Constructing and examining several scenarios for design decisions that are made several times a day is not feasible. Leaving these decisions to training and skill is not optimal either because acquiring these skills is more complicated and time-consuming than it is to learn how to make trivial decisions. Furthermore communicating the reasons for preferring one solution over another one in such a case may be difficult if the decision is simply based on "experience". So a light-weight approach is needed to support these non-trivial daily design decisions.

2.1.3 Communicating Reasons

When doing design, it is necessary to communicate with other developers. Reasons for design decisions have to be explained and advantages and drawbacks have to be discussed. When a team is doing design, they have to communicate among each other and when there are separate designers and programmers, these two groups also have to communicate. Moreover experienced designers have to communicate with novices.

Communication is important but it is not easy. A common vocabulary is necessary and a common understanding of design. Especially when experienced designers talk to novices they might be tempted to just claim that a certain design decision is just based on their experience. This may be true but in such a case the novice has no chance to really understand the decision.

Principle languages also help here. They form a common vocabulary of design so advantages and disadvantages can be communicated. Novices can learn the principle language as a language for talking about design.

2.2 Using Principles for Design

2.2.1 Origin of Principles

Human learning comprises the inference of rules. When certain observations or experiences are made, a rule may be proposed which helps making predictions about future observations.

Such rules may be definite laws or just rules of thumb, heuristics or approximations. The establishment of laws is difficult and time-consuming. On the other hand simple rules of thumb don't need to be true in every case. So they can be established much easier as a formal proof is not necessary and a sound reason suffices. They just need to be helpful.

Sometimes these rules are just intuitively used and not expressed in any way, thus staying tacit knowledge. Tacit knowledge is only helpful for those who have it but cannot be communicated to others. So in order to teach experiences, tacit knowledge needs to be made explicit. Experienced software designers wrote about the experiences they made and the rules they learned. By stating the rules as memorable principles, they made their tacit knowledge explicit and teachable.

Some of these principles are very old as they have been discovered very early in the history of computer science. Principles like *separations of concerns*, *low coupling*, *high cohesion* or *goto statement considered harmful* date back to the 60ies and 70ies. Other principles are rather new and sometimes only known to a specific community. And moreover some knowledge still remains tacit and is not yet expressed as a principle although this would be possible.

This thesis addresses principles for object-oriented general-purpose software design. Apart from that, there are also principles for other contexts like framework design, user interface design, coding, high-level architecture, or development process organization. The idea is rather general and can be applied in a variety of contexts.

2.2.2 Conflicting Principles and Trade-Offs

Each principle describes a certain aspect of the problem. For example →4.3.2 *Keep It Simple Stupid* (KISS) is about the value of simplicity. A solution is better when it is simpler. Another principle that might also be considered in the same context is the →4.3.5 *Generalization Principle* (GP) which says that a more generic solution is better than a specific one, as it can be applied to a broader set of problems which increases reusability.

This is a typical example of two conflicting principles. Both are valid. A solution is better (w. r. t. ease of writing, ease of use, readability and potential for fault introduction) when it is simpler. And a solution is better (w. r. t. reusability and changeability) when it is more general. These principles are conflicting. A solution normally is either simple or generally applicable but not both. As the aspects, the principles refer to, are typically not binary, adherence to the principles can be informally rated on a gradual scale. A good design decision will now balance these principles and come up with a compromise that is generalized to some extend but not too much in order to keep the solution reasonably simple.

While typically there is no solution which is good in every aspect or principle (e. g. which is totally simple *and* totally generic) there might be solutions that are bad in both aspects/principles (e. g. ones which are neither simple nor generally applicable). So a good solution is "Pareto-optimal", which means there is no other solution which is better in one aspect/principle while being at least equally good in all others.

In that way good solutions can be distinguished from bad ones. Nevertheless there can be several good, i. e. Pareto-optimal, solutions. For choosing between them, it is necessary

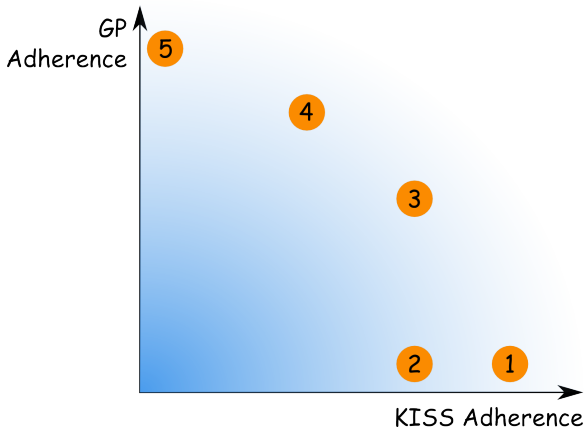


Figure 2.1: Possible solutions for a problem and their adherence to GP and KISS principle.

to make a judgment which aspect is more important. When readability is more important, the simpler solution should be chosen and in case of reusability being more important the generic one should be chosen.

2.2.3 Example

A trivial example that shows the idea could be a square root function. In a program that needs the square root of 2, the following design problem could arise: *What is the best way to compute the square root of 2?* There are several possible solutions:

1. A constant: `sqrt_2 = 1.4142135623730951 /* approx. */`
2. A method that computes the square root of 2: `double sqrt_2()`
3. A method that computes arbitrary real square roots of real numbers:
   `double sqrt(double radicand)`
4. A method that computes arbitrary real powers to real numbers:
   `double power(double base, double exponent)`
5. A method that computes arbitrary complex powers to complex numbers to an arbitrary precision:
   `BigComplex power(BigComplex base, BigComplex exponent, BigDecimal epsilon)`

Figure 2.1 depicts the solution space of the problem. Possible solutions lie in the blue area. If it existed, an optimal solution with respect to both principles, KISS and GP, would be shown in the top right corner. And an overly complex solution which is not general at all would be near the origin. The above solutions are numbered in the graphic. Note that this is just an informal rating which can be done quickly and doesn't involve complex metrics.

Each solution has its advantages and disadvantages. Solution 1 is the easiest. It is simple to implement and easy to read. On the other hand it is very specific and can only be reused when exactly the same value is needed. This is the solution that follows KISS to the greatest extend. It is very simple.

In contrast to that, solution 5 is very generic and can be applied to a broad set of problems. So it can be reused in many other contexts. The downside is that it is difficult to write and difficult to use. These are the both extremes when one principle is almost neglected.

Solution 2 represents a bad solution. It is certainly more complex than 1 but does not provide a more general solution. And it is also less general than 3 but is not simpler. Figure 2.1 shows the same *GP adherence* for solutions 1 and 2 and the same *KISS adherence* for solutions 3 and 2. Thus the inferior solution 2 is shown deeper in the blue area near the origin.

The solutions that are commonly implemented in standard math libraries are 3 and 4. They represent a compromise between simplicity and general applicability. Neither is better than the other. 4 is more complex but also more powerful. They both represent Pareto-optimal solutions.

And even solutions 1 and 5 are Pareto-optimal. They may be less common but one could think of situations when they might be preferred over the others. There may be a case when the square root of 2 is the only root necessary and a math library containing a square root function may not be available due to limited memory on a special purpose hardware. Or there may be complex numbers necessary for a certain scientific computation.

This approach does not tell which of the four good solutions should be chosen (only solution 2 is sorted out). It is more a way of thinking than a general method. Rating solutions with real numbers or even visualizing the solution space graphically (like in figure 2.1) is not necessary and in most cases also not helpful. But stating that the design decision to make is a question of simplicity versus general applicability is a valuable statement which helps the designer to find an appropriate solution. The approach is about exploring the dimensions of the solution space to a given design problem.

2.2.4 Principle Languages

There are not only two but many principles and thus aspects or effects to consider when designing software. The design space is multi-dimensional. Principle languages interconnect principles in a way that the consideration of one principle inevitably leads to other principles to consider. So a principle language helps finding relevant principles for a given design problem.

The principle discovery starts with one or two starting principles. These starting principles are typically obvious from the given design problem. For these principles the principle language lists related principles which are likely to be relevant, too. By repeatedly looking for related principles a set of characterizing principles is found. As this is meant to be a lightweight technique, there are not many prescriptions for how to do principle discovery. The principle discovery process is neither determined, nor deterministic. The order in which the principles are scanned for their relations is not defined. Principle discovery can be

aborted anytime at the decision of the designer. When the designer decides that enough principles have been found to make an informed decision, there is no need to continue principle discovery. So the characterizing set is also not unique. The idea is to help the designer; not to impose any unnecessary work.

The characterizing set of principles describes the relevant aspects to consider when deciding upon the solution for the given design problem. The designer can then informally rate the possible solutions according to the design principles in the characterizing set. Using this information the designer can make a sound and comprehensible design decision. It is also easy to explain the reasons which lead to the choice because they just lie in the principles.

Again this is meant as a guidance not as an obligation. The whole principle discovery, rating, and decision making typically does not take more than a few seconds and in complicated cases maybe minutes.

The idea of supporting design is similar to patterns. Principles and patterns, principle languages and pattern languages aim at giving guidance to designers by codifying experience. But in contrast to principle languages, pattern languages are normally not learned, but rather used as a kind of handbook. There is too much detail in patterns (especially the structure and the different possibilities for variations) which makes memorizing all the details infeasible.

Principles on the other hand are much simpler. They can be easily memorized as the core idea of each principle can be stated in one or two sentences. Nevertheless designers who are new to the principle language can use it as a handbook, too. Over the time the principle language will be learned by the designers so looking up the principles is not necessary anymore. The principle language then becomes a real “language”, meaning a common set of vocabulary that can be used to talk about designs.

2.3 Relation to Other Approaches and Other Research

2.3.1 Design Patterns

There are certain similarities between principles and patterns. As already described in section 1.2 both are forms of experience reuse but patterns reuse solution schemes whereas principles reuse judgments upon solutions. Design principles are rules of thumb helping to make a design decision and design patterns are solution templates that can be applied to a problem.

Design patterns are not a complete generative design approach, as there is no complete general-purpose pattern language for software design. This means patterns are rather certain design tricks which are useful in very particular situations and not an approach generating a complete design from a set of requirements. So patterns do not replace approaches doing so. Nevertheless patterns in contrast to principles help generating designs.

Furthermore a judgment is needed on whether to apply a pattern or not. Applying patterns imposes some drawbacks (typically increased complexity) and patterns can be applied in different ways and variations. Moreover an overuse of patterns is harmful and results in bad design. Peter Sommerlad one of the authors of “Pattern-Oriented Software Architecture” [4]

writes in [7]: "Whenever designers feel the itch to apply a design pattern, I would ask them first to think. Do they really need its flexibility, and will adding the pattern's complexity make the overall system simpler? In addition, I'd ask them to [...] relearn, understand, and apply the basic principles of good design such as simplicity, low coupling, and high cohesion."

So an analytic design approach like the one proposed here is needed. Patterns and principles complement each other. Patterns can give solution schemes and principles help judging how and whether to apply them.

Principles and patterns are complements. But there are also some other views on this relationship. First of all design patterns are said to balance "forces" with a force being any aspect that influences the solution of the given design problem [4]. Principles can be seen as such forces. They influence the design decision, they drag the design in different directions, and they need to be balanced in order to find a suitable compromise.

And another view is to see principles as reasoning patterns. The term pattern can be applied widely. They are proven solution schemes to recurring problems in a certain context. And in the context of making design decisions there are proven reasoning schemes used for judging designs. These reasoning schemes are design principles.

2.3.2 Refactorings and Code Smells

Principles also have some similarities with refactorings and code smells. In his book "Refactoring: Improving the Design of Existing Code" [8] Martin Fowler describes 72 refactorings, i. e. procedures that transform bad code to good code. Such refactorings are for example "rename method", "move method", or "replace method with method object". These refactorings are applied whenever the code "smells", which means it has some characteristic that marks it bad code. Fowler describes 22 of these code smells. Examples for code smells are "long method", "long parameter list", and "message chains". Each code smell is associated with a set of refactorings which cure it.

Each of the smells can be seen as a principle when "avoid" is prepended: "avoid long methods", "avoid long parameter lists", "avoid message chains", etc. These are fine-grained principles dealing with a very specific aspects. Some of them are code-centric and are rather concerned with coding and less with design (for example "(avoid) switch statements") but others are clearly about design (e. g. "avoid speculative generality").

There are also similarities from a process point of view: Code smells tell if a solution is bad. Principle languages give a kind of reasoning framework to tell good solutions from bad solutions. So a principle language can be used instead of code smells in order to find possibilities for design improvement and to initiate refactorings.

2.3.3 Existing Principle Collections

As already stated, principles as such are not new. There are also already principle collections sometimes even principle catalogs which means, there is a consistent description template for the principles of the collection (see definition 7).

The best known principle collection are probably Robert C. Martin's SOLID principles. In [9] he describes the fife principles SRP (Single Responsibility Principle), OCP (Open-Closed Principle), LSP (Liskov Substitution Principle), ISP (Interface Segregation Principle) and DIP (Dependency Inversion Principle). For each of these principles there is a separate chapter comprising an in-depth description and examples. All these principles are principles for object-oriented design. They are broadly applicable and because of this and other books and articles (e. g. [10], [11]) well known in practice. Later in the book he adds fife further principles which are concerned with more coarse-grained package structures.

Martin's principles are not or only loosely interconnected so his collection is not a principle language. The principles are described in isolation and also meant to be applied in isolation. The whole approach taken here is different. In Martin's view principles are similar to refactorings. They heal what he calls "design smells". A design smell is an unintended property of a software design for example *needless complexity* (i. e. overdesign), *fragility* (the design is easy to break), or *immobility* (the design is hard to reuse) [9, p. 85].

Design smells are very general and difficult to apply as principles in the sense discussed here. They are rather indications that something is wrong with the design and refactoring is needed. Martin then uses his principles generatively as a kind of refactoring to improve the design and thus remove the design smell.

Another principle collection is described in Craig Larman's "Applying UML and Patterns" [12]. Larman describes what he calls General Responsibility Assignment Software Patterns or GRASP for short. Fife of these nine "patterns" are rather principles: Information Expert, High Cohesion, Low Coupling, Creator, and Controller. These principles are described using a fixed description template (so GRASP is a principle catalog) and they are even loosely interconnected in the sense that related principles (or "patters" as they are called in the book) are listed.

Similarly to Robert C. Martin, Larman uses the principles rather generatively than analytically. Only for low coupling an high cohesion Larman notes "Use this principle to evaluate alternatives" [12, pp. 299, 314]. The idea behind GRASP is to help assigning responsibilities to classes. This is a major part of OOD but not everything.

In "The Pragmatic Programmer" [13] Andrew Hunt and David Thomas give 70 "tips" for software development, sixteen of which are principles. And nine of these sixteen principles are concerned with software design. There is no description template and only loose interconnection by chapter references. Software design is also not the primary topic of the book so there is no advice on how to use the tips for doing design. There is neither an analytic, nor a generative design approach but rather the 70 tips are just meant as a concise summary of the book.

Bertrand Meyer describes 200 principles, rules, precepts and definitions in his book "Object-Oriented Software Construction" [14]. But only fourteen of them are principles in the sense discussed here. Meyer mainly describes concepts and usage of his programming language Eiffel. His book only deals with object-oriented design as this task influences and is influenced by language constructs. This is the reason why only seven percent of his rules are relevant to this thesis. There is no description template, no principle relationships and no description of an approach dealing with the application of principles.

In “The Art of Unix Programming” [15] Eric S. Raymond describes the Unix philosophy using 17 principles. Also here is no description template, no relationships and no principle-based design approach.

Lastly there is also a book about software development principles: “201 Principles of Software Development” by Alan M. Davis [16]. 26 of the 201 “principles” are about software design and half of them are principles according to the definition used here. The book is mainly a large list and contains only one short explanation on every principle.

After the principle language has been presented in chapter 4, section 5.3 compares these principle collections to the presented language.

Summarizing this, there are plenty of principles known and there are also a few principle collections. But often the focus is just to collect arbitrary software development knowledge. The principles are not or only insufficiently interconnected and with the exception of SOLID there is no guidance on how to use the principles for software design. There is nothing comparable to a principle language, yet. Also SOLID, which already gives some guidance, is not a principle language as it takes a different approach and is applicable to fewer design problems as there are only fife principles.

2.3.4 Laws and Theories Forming a Body of Knowledge

The most systematic principle collection so far can be found in [17]. Endres and Rombach discuss laws and theories as a body of knowledge for software engineering. Their goal was to collect empirically verified knowledge about software and systems engineering independent of any particular engineering approach, technology or way of thinking. This forms a body of knowledge for practitioners and catalogs existing findings for researchers.

There is a rough classification of findings into laws, hypotheses and conjectures. For laws there is strong evidence such that they can be assumed true without further need for research. Hypotheses on the other hand are only tentatively accepted. There may be some evidence and there may also be some doubt. Lastly conjectures are pure guesses where science is yet to examine the effect. This categorization is done for two reasons: Firstly practitioners can use it to judge how much to trust the findings. Laws can be assumed true whereas hypotheses and conjectures need to be taken with care. And secondly researchers can use the categorization to focus their research on what is still unknown.

For each law (but not for hypotheses and conjectures) a theory is listed which tries to explain why the law is valid. This is similar to the rationale a principle description gives. The purpose is to give an understandable reason for the law.

The work is primary literature research and does not present a particular design approach, a relationship between the laws or anything else that resembles a principle language. Nevertheless it is a helpful source for principles. The book lists rules for all aspects of software engineering, including tasks like verification and requirements definition. Counting only those which are relevant for software design Endres and Rombach list sixteen laws, eight hypotheses and two conjectures. These are the ones concerned with system design, composition, and maintenance, i. e. those tasks where design decisions are made.

But not all of these are principles. Some of these rules are rather phenomenological

descriptions of effects that need no consideration while making design decisions (e. g. Conway's Law which states that a "system reflects the organizational structure that built it" [17, pp. 81 f.]). Others have a different focus—often management and process planning (e. g. Lanergan's Law which sates that the "larger and more decentralized an organization, the more likely it is that it has reuse potential" [17, p. 76])—or are otherwise not helpful for the tasks discussed here. Only six laws and two hypotheses could be used as or at least modified to software design principles.

So Endres and Rombach have a different focus. They concentrate on empirical observations whereas this thesis is concerned with principles which directly help developers make and communicate design decisions.

A comparison between the principles collected by Endres and Rombach and the principle language presented in chapter 4 is given in section 5.3.

3 Towards a Principle Language

3.1 Using a Wiki for Describing Principles

Along this thesis a wiki was used to describe the principles discussed here as well as several others. A wiki is an ideal tool for describing principles. This is because the very first wiki, the origin of all wikis, was created by Ward Cunningham as a tool for describing patterns. "The Wiki" is the home of the Portland Pattern Repository [18]. Later the use of wikis became more wide-spread. But as the primary idea was to describe and discuss patterns, it is only natural to use the same tool for describing and discussing principles.

One advantage of wikis is, that they make it easy to link pages so a hypertext is created. Principle languages precisely are about linking principles and wikis simplify navigating their web structure. Additionally they can be used to collaboratively describe further principles, develop further principle languages, and improve existing ones.

One of the goals of this work is, to construct a general-purpose principle language for object-oriented design. Chapter 4 shows the result. This is an except from the wiki which includes those principles that are part of the principle language. Beside them there are several supplementary principles which are also described and linked in the wiki but which are not directly part of the principle language.

3.2 The Principle Description Template

In order to construct and document a principle language, it has to be defined, how the single principles should be described. As a principle language is a principle catalog (see definitions 7 and 8), principles should be described using the same structure. This section describes how principles are documented in this thesis. The wiki contains more sections since it is intended to use the wiki on a broader scale. Eventually it will include principles in other contexts, other principle languages, a glossary, and maybe also pattern descriptions. This section only deals with the sections relevant to the thesis. The wiki has its own page explaining the broader wiki template.

3.2.1 Variants and Alternative Names

Each principle may have several alternative names. This may be because the same principle has been described several times independently. A principle may also evolve over time, change its name, change its meaning, may be applied to other contexts, etc. So there may be several names referring basically to the same principle. This also means that the alternative names may roughly correspond to certain views on the principle. The views may differ

slightly resulting in certain variations of the principle. Alternative names are listed in this section and, if necessary, explained.

3.2.2 Principle Statement

The principle statement is basically a concise explanation in one or two sentences. This may be the original wording or a new one. The main purpose is to give a memorable "definition" of the principle.

3.2.3 Description

As one or two sentences are never enough to explain a principle in detail, there is a separate section describing what the principle means and how it is applied.

3.2.4 Rationale

Principles normally are not hard rules but rather heuristics or rules of thumb. So there is no formal proof showing that the principle is correct in each and every situation. Nevertheless there needs to be a reason for the principle, meaning some rationale explaining why it is valid. In order to assess whether the principle is applicable to a certain problem, the rationale can be used. If the reasons given in this section apply to the given problem, the principle can be applied.

3.2.5 Strategies

The main idea behind principles is to assess solutions and not to construct solutions. Nevertheless applying principles may lead to the conclusion that a solution is not as good as it could be. This section lists strategies that can be used to transform a given solution in a way that the result better adheres to the principle. So the principle language provides guidance beyond the pure assessment of design solutions. A generative approach for constructing solutions (see section 2.1.1) is nevertheless necessary.

3.2.6 Caveats

This section lists warnings on how *not* to use this principle. Disadvantages are partly treated below in the section "contrary principles". Nevertheless there are sometimes helpful remarks on pitfalls and wrong usages that cannot be described using contrary principles or that are otherwise noteworthy. This section discusses these issues.

3.2.7 Origin

This section describes where the principle comes from, where it has been prominently described, etc.

3.2.8 Evidence

Apart from the rationale there may be different evidence that the principle is valid:

Proposed If the principle is neither *examined*, nor *accepted* it is marked *proposed*.

Examined A principle is marked *examined* if and only if it has been subject to scientific research showing evidence beyond constructed examples. Note that this state is called "examined" and not "proven" as the examination may have limitations and the principle may still be *questioned*. If the scientific examination supports the principle, it may be considered state of the art.

Accepted A principle is marked *accepted* if and only if it is widely used in practice. This is assumed if there is a publication widely known to practitioners which describes the principle. So an *accepted* principle can be regarded state of the practice.

Examined and Accepted A principle may be both *examined* and *accepted*.

Questioned Independent of whether a principle is *proposed*, *examined*, *accepted* or *examined* and *accepted*, a principle may also be *questioned*. A principle is *questioned* if there are comprehensible reasons that the principle may be wrong or misleading. Note that the principle itself needs to be questioned. It is not sufficient that keeping the rule may inevitably also have some negative effects expressed by another principle. This is normal due to the nature of the principle definition used here. A principle is *questioned* if there is doubt concerning the positive effect it claims to have.

3.2.9 Relation to Other Principles

There are certain relationships among principles. This section lists and explains them so the consideration of one principle inevitably leads to other principles that can be considered. An important aspect of this is that the pure purpose of this list of relationships is to give a clear navigation path to the principles that should be considered next. These relationships are fuzzy and sometimes not valid in every respect. But this is not a problem since their purpose is to be practically useful. A purely "academic" analysis of the principles which ignores their practical usage is not intended.

Generalizations

A generalization of a principle is another principle that can be applied in a broader context. Generalizations may be considered in addition or instead of the principle. If a principle does not fit well to the problem, a generalization of it may fit better. It may also add further reasons and a broader view for the assessment of possible solutions.

The principle is always a specialization of its generalizations. There may also be several generalizations, for example when a principle is a consequence of two others.

Specializations

A specialization is a more concrete principle that either applies in a narrower context or is some kind of corollary. Specializations may be considered in addition or instead of the principle. If a principle does not fit well to the problem, a specialization of it may fit better. It may be easier to apply and more tailored to specific kinds of problems.

The principle is always a generalization of its specializations and naturally there may be several specializations.

Contrary Principles

Following the principle may have a negative impact on aspects addressed by other principles. These contrary principles are listed here and the consequence is explained. Like there is no guarantee that the principle itself is valid in every case, there is also no guarantee that there is the negative effect concerning the other principles. There is just a high probability that there is this effect. Therefore these principles should also be considered if this one is applied.

As the relationships are purely for navigational purposes, the "is-contrary-to" relationship is not necessarily symmetric.

Complementary Principles

A principle is always a reduction of the given design problem to a very specific aspect or effect. Other principles have to be considered too in order to have a full picture of the design problem. Sometimes when one principle is considered, another one is very likely to be relevant too despite not being contrary. This is then a complementary principle. As for the other relations this is just a tendency and a purely navigational relationship. In practice a complementary principle may also be contrary or not applicable.

Similar to "is-contrary-to", the "is-complementary-to" relationship is not necessarily symmetric.

3.2.10 Examples

One or more self-contained examples explains how the principle distinguishes "good" and "bad" solutions with respect to the aspect the principle is about or exemplifies certain relationships, strategies, caveats, etc.

3.3 Constructing a Principle Language

There are tens and maybe hundreds of principles. In order to form a concise vocabulary of principles for communicating about software design a manageable subset is needed. Such a subset forming a principle language can be taught and learned more easily. But constructing such a principle language comprises some complicated tasks and considerations.

A principle language should cover most of the arising design problems in the context it is built for. For this thesis a principle language for object-oriented design (OOD) has been

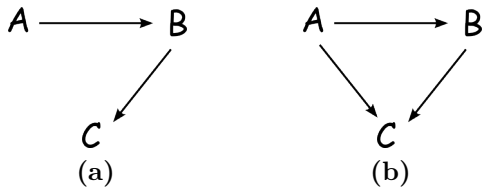


Figure 3.1: Transitive relationships: (a) without (b) with shortcut.

constructed. So in this case most of the design problems arising when doing OOD should be covered. In general this means that the principle language has be be large enough to have this kind of coverage so but otherwise as small as possible so it is still learnable.

In order to find a suitable compromise a tradeoff between granularity and utility has to be made. Michael Jackson writes on methods: "It's a good rule of thumb that the value of a method is inversely proportional to its generality." [19] The same can be said about principles: Very general principles are broadly applicable to a large variety of problems. But applying them can be difficult. The more precise a principle is the more helpful it will be regarding a certain class of problems. But this also means that a specialized principle only applies to a fewer problems.

Finding the right set of principles is very important but the most complex task when creating a principle language is finding the correct, meaning the most helpful, relationships. The principle language which is presented in chapter 4 comprises 19 principles but 93 relationships. And even very early versions of the principle language contained almost the same principles as now. Only three principles changed. On the other hand about half of the relationships changed over time, some even several times.

Constructing the principle language requires considering every possible relationship between the given set of principles and answering the following question: When considering principle A, do I also want to consider principle B? It is not enough to look at the principles themselves but it has to be envisioned in which situations one might consider principle A and wants to navigate to principle B.

This can be considered one of the key findings of this thesis: When a principle language is to be constructed, not the actual mathematically-clear relationships (if there are any) are important. The important aspect about the relationships is how they are used.

Moreover transitivity has to be considered in some special way: Navigating the principle language is to some extend transitive. Suppose there are the principles A, B and C. Considering A may lead to B and considering B may lead to C. In such a case there is the question whether or not to create a "shortcut-relationship" directly going from A to C. See figure 3.1 for illustration. The difference is that in the first case C is only considered when B is considered. In the second case the shortcut creates a direct navigation path from A to C. So C can be considered without having B. Here is also the question how a developer would want to navigate the principle language. It depends on the possible design decisions and not merely on the principles.

Next the type of the relationship has to be determined. It can be generalization/special-

ization, is-contrary-to or is-complementary-to. This is also not clear by just looking at the principles. A principle which is contrary in one situation may be complementary in another one. So it has to be judged which relationship type reflects the connection best.

Another possibility would be not to use relationship types and just use generic "may-navigate-to" relationships. So the relationship would just be a "see also" similar to the currently used is-complementary-to. This is possible but some information is lost. Generalization/specialization relationships indicate that the related principle can replace the currently considered one. And is-contrary-to relationships are generally more important than others as they point to possible drawbacks. For these reasons relationship types are used in the principle language created for this thesis.

4 A Principle Language for Object-Oriented Design

4.1 Overview

In the following a principle language for object-oriented design will be presented. Its purpose is to help the designer make sound design decisions. This comprises the choice upon how to modularize the software, which responsibilities to assign to these modules, and how modules should interact with each other. It is not the purpose of the principle language to help designing algorithms, to structure requirements, to do user interface design, etc. The principle language also aims to be used with the object-oriented programming paradigm. For other purposes, contexts and programming paradigms, other sets of principles and other principle languages would be necessary although some of the principles described here could also be part of those principle languages.

The principle language consists of 19 principles which can roughly be grouped into four categories: general principles, modularization principles, module communication principles, interface design principles and internal module design principles. Table 4.1 in page 25 lists the principles and figure 4.1 shows the relationships between them.

4.2 The Wiki

All the principles have been described in a wiki. The wiki is designed much more general than this thesis as it is planned to continue using and working on it. In the future it will contain principles on other levels of abstraction, non-principles (rules which look similar to principles but are not), a glossary, and eventually maybe also patterns and anti-patterns. It already contains several principles which haven't become part of the language. Because of this broader design of the wiki, it also contains some more sections than the principle description template showed in section 3.2 and also more references to related papers, books, and online resources. The goal of the wiki is to create a platform for collaborative work on principles and principle languages. It will go online shortly after this thesis has been handed in.

The following sections describe the principles of the principle language in more detail. This is a snapshot from the wiki reduced to those principles which became part of the language. The principle descriptions vary in detail and some sections lack content. These empty sections are not removed here in order to show that this is a part of a wiki which continues in development. Research on principle languages is by far not at its end. It has just started and there are still plenty of research possibilities (see chapter 7). The wiki is designed to stay a part of that development.

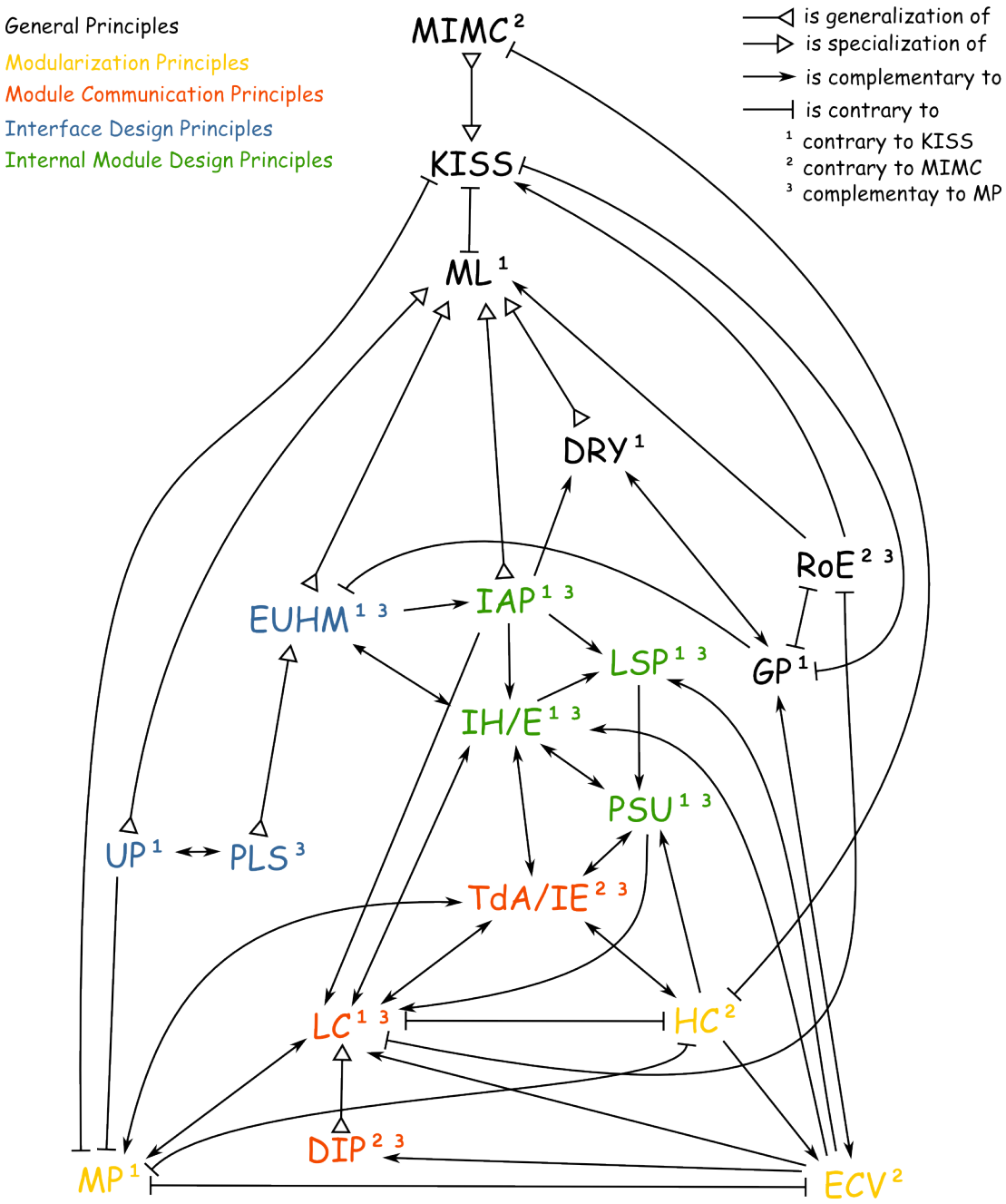


Figure 4.1: Overview showing the principle language

Table 4.1: The Principles of the Principle Language

4.3 General Principles

4.3.1 Murphy's Law (ML)

Variants and Alternative Names

- Design for Errors[16]

Principle Statement

Whatever can go wrong, will go wrong. So a solution is better the less possibilities there are for something to go wrong.

Description

Although often cited like that, Murphy's Law actually is not a fatalistic comment stating "that life is unfair". Rather it is (or at least can be seen as) an engineering advice to design everything in a way that avoids wrong usage. This applies to everything that is engineered in some way and in particular also to all kinds of modules, (user) interfaces and systems.

Ideally an incorrect usage is strictly impossible. For example this is the case when the compiler will stop with an error if a certain mistake is made. And in case of user interface design, a design is better when the user cannot make incorrect inputs as the given controls won't let him.

It is not always possible to design a system in such a way. But as systems are built and used by humans, one should strive for such "fool-proof" designs.

There are different kinds of possible errors that can and according to ML eventually will occur in some way: Replicated data can get out of sync, invariants can be broken, preconditions can be violated, interfaces can be misunderstood, parameters can be given in the wrong order, typos can occur, values can be mixed up, etc.

Note that Murphy's law also applies to every chunk of code. According to the law the programmer will make mistakes while implementing the system. So it is better to implement a simple design, as this will have fewer possibilities to make implementation mistakes. Furthermore code is maintained. Bug-fixes will be necessary and present functionality will be changed and enhanced so every piece of code will potentially be touched in future. Hence a design is better the fewer possibilities there are to introduce faults while doing maintenance work.

Rationale

Systems are built and used by humans. And as humans always will make mistakes, there always will be some possibilities for a certain mistake. So if some mistake is possible, eventually there will be someone who makes this mistake. This applies likewise to system design, implementation, verification, maintenance and use as all these tasks are (partly) carried out by humans.

This means the fewer possibilities there are that a mistake is made, the fewer there will be. As mistakes are generally undesirable, a design is better when there are fewer possibilities for something to go wrong.

Note that ML does *not* claim that everything constantly fails unless there is no possibility to do so. It simply says that statistically in the long run a system will fail if it can.

Strategies

This is a very general principle so there is a large variety of possible strategies to adhere more to this principle largely depending on the given design problem:

- Make use of static typing, so the compiler will report faults

- Make the design simple, so there will be fewer implementation defects (see →4.3.2 *Keep It Simple Stupid* (KISS))
- Use automatic testing to find defects
- Avoid duplication and manual tasks, so necessary changes are not forgotten (see →4.3.4 *Don't Repeat Yourself* (DRY))
- Use polymorphism instead of repeated switch statements
- Use the same mechanisms wherever reasonably possible (see →4.6.3 *Uniformity Principle* (UP))
- Use consistent naming and models throughout the design (see →4.4.1 *Model Principle* (MP))
- Avoid preconditions and invariants (see →4.7.2 *Invariant Avoidance Principle* (IAP))
- Use assertions to detect problems early
- ...

Caveats

See section contrary principles.

Origin

The exact wording and who exactly coined the term, remains unknown. Nevertheless it can be stated that its origin is an experiment with a rocket sled conducted by Edward A. Murphy and John Paul Stapp. During this experiment some sensors have been wired incorrectly. A more accurate quote might read something like this: "If there's more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way." A detailed version of the history of the experiment and the law can be found in [20].

Evidence

- Accepted The principle is widely known and it's validity is assumed. See for example the Jargon File [21]. Nevertheless sometimes it is rather used as a kind of joke instead of an design advice.

Furthermore every defect in any system is a manifestation of ML. If there is a fault then obviously something went wrong. The correlation between the number of possibilities for introducing defects and the actual defect count can be regarded trivially intuitive.

Relations to Other Principles

Generalizations

Specializations

- →4.3.4 *Don't Repeat Yourself* (DRY): Duplication is a typical example for error possibilities. In case of a change, all instances of a duplicated piece of information have to be changed accordingly. So there is always the possibility to forget to change one of the duplicates. DRY is the application of ML to duplication.
- →4.6.1 *Easy to Use and Hard to Misuse* (EUHM): Because of ML an interface should be crafted so it is easy to use and hard to misuse. EUHM is the application of ML to interfaces.
- →4.6.3 *Uniformity Principle* (UP): A typical source of mistakes are differences. If similar things work similarly, they are more understandable. But if there are subtle differences in how things work, it is likely that someone will make the mistake to mix this up.
- →4.7.2 *Invariant Avoidance Principle* (IAP): Invariants are statements that have to be true in order to keep a module in a consistent state. ML states that eventually an invariant will be broken resulting in a hard to detect defect. IAP states that invariants should therefore be avoided. So IAP is the application of ML to invariants.

Contrary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): On the one hand a simpler design is less prone to implementation errors. In this aspect KISS is similar to ML. On the other hand it is sometimes more complicated to make a design "fool-proof" so usage and maintenance mistakes are prevented. In this aspect KISS is rather a contrary principle. Both applies at the same time so a tradeoff has to be made whether correct implementation or correct usage and maintenance are more important in the given case. This means, it is necessary to consider KISS in addition to ML in order to find a suitable compromise. See example 1: parameters.

Complementary Principles

Examples

Example 1: Parameters Suppose there are two methods of a string class `replaceFirst()` and `replaceAll()` which replace the first or all occurrences of a certain substring, respectively.

The following method signatures are a bad choice:

```
replaceFirst(String pattern, String replacement)
replaceAll(String replacement, String pattern)
```

Eventually someone will mix up the order of the parameters leading to a fault in the software which is not detectable by the compiler.

So it is better to make parameter lists consistent:

```
replaceFirst(String pattern, String replacement)
replaceAll(String pattern, String replacement)
```

This is less error prone. When for example a call to `replaceFirst()` is replaced by a call to `replaceAll()`, one cannot forget to exchange the parameters anymore. This is how is is done in the Java API [22].

But here still one could mix up the two string parameters. Although this is less likely, as having the substring to look for first is "natural", such a mistake is still possible. An alternative would be the following:

```
replaceFirst(Pattern pattern, String replacement)
replaceAll(Pattern pattern, String replacement)
```

Here both methods expect a `Pattern` object instead of a regular expression expressed in a string. Mixing up the parameters is impossible in this case as the compiler would report that error. On the other hand using these methods becomes a bit more complicated:

```
"This are a test.".replaceFirst(new Pattern("are"), "is");
```

[1] instead of

```
"This are a test.".replaceFirst("are", "is");
```

→4.3.2 *Keep It Simple Stupid* (KISS) is about this disadvantage.

Example 2: Casts and Generics Another example for the application of Murphy's Law would be the avoidance of typecasts:

```
List l = new ArrayList();
l.add(5);
return (Integer)l.get(0) * 3;
```

This works but it makes a cast necessary and every cast circumvents type checking by the compiler. This means it is theoretically possible that during maintenance someone will make a mistake and store a value other than Integer in the list:

```
l.add("7");
```

Murphy's Law claims that however unlikely such a mistake might seem, eventually someone will make it. So it is better to avoid it. In this case this could be done using Generics:

```
List<Integer> l = new ArrayList<Integer>();
l.add(5);
return l.get(0) * 3;
```

1 Note that in the Java API it would rather be `Pattern.compile()` instead of `new Pattern()`; see [23]

Here this mistake is impossible as the compiler only allows storing integers.

Note that the typecast is rather a symptom than the actual problem here. The problem is, that the `List interface` is not generic and the symptom is the typecast. The reason for this flaw is, that the `List interface` predates the introduction of generics in Java.

Example 3: Date, Mutability/Aliasing In Java the classes `Date` [24] as well as the newer `Calendar` [25] are mutable which means the reference semantics of Java objects may cause unintended alternations of date values. Eventually someone will copy the reference to a date object instead of copying the object itself, which is usually a mistake when programming with dates.

```
Date date1 = new Date(2013, 01, 16);
Date date2 = date1;
System.out.println(date1); // Sun Feb 16 00:00:00 CET 3913
System.out.println(date2); // Sun Feb 16 00:00:00 CET 3913
date1.setMonth(2);
System.out.println(date1); // Sun Mar 16 00:00:00 CET 3913
System.out.println(date2); // Sun Mar 16 00:00:00 CET 3913
```

Furthermore as can be seen in the code above, the month value counterintuitively is zero-based, which results in 1 meaning February. This obviously is another source for mistakes.

Because of these and several other flaws in the design of the Java date API, most of the methods in `Date` are deprecated and also the newer `Calendar` API will be replaced by a new API [26] in Java 8.

4.3.2 Keep It Simple Stupid (KISS)

Variations and Alternative Names

- (Rule of) Simplicity
- KISS may also mean “Keep it short and simple”, “keep it simple and straightforward”, “keep it smart and simple”, etc. A large amount of variations exists.

Remarks: “Stupid” may be interpreted as an adjective or a noun. Compare the two variants “keep it simple and stupid” vs. “keep it simple, stupid!”. Despite all these alternative names the general idea of the KISS principle is always the same.

Principle Statement

A simple solution is better than a complex one—even if the solution looks stupid.

Description

The KISS principle is about striving for simplicity. Modern programming languages, frameworks and APIs have powerful means to create sophisticated solutions for various kinds of

problems. Sometimes developers might feel tempted to write "clever" solutions that use all these complex features. The KISS principle states that a solution is better when is uses less inheritance, less polymorphism, fewer classes, etc.

A solution that follows the KISS principle might look boring or even "stupid" but simple and understandable. The KISS principle states that there is no value in a solution being "clever" but in one being easily understandable.

This does not mean that features like inheritance and polymorphism should not be used at all. Rather they should only be used when they are necessary or there is some substantial advantage in using them.

Rationale

A simpler solution is better than a complex one because simple solutions are easier to maintain. This includes increased readability, understandability, and changeability. Furthermore writing simple code is less error prone.

The advantage of simplicity is even bigger when the person who maintains the software is not the one who once wrote it. The maintainer might also be less familiar with sophisticated programming language features. So simple and stupid programs are easier to maintain because the maintainer needs less time to understand them and is less likely to introduce further defects.

Strategies

This is a very general principle so there is a large variety of possible strategies to adhere more to this principle largely depending on the given design problem:

- Avoid inheritance, polymorphism, dynamic binding and other complicated OOP concepts. Use delegation and simple if-constructs instead.
- Avoid low-level optimization of algorithms especially when involving Assembler, bit-operations, and pointers. Slower implementations will work just fine.
- Use simple brute-force solutions instead of complicated algorithms. Slower algorithms will work in the first place.
- Avoid numerous classes and methods as well as large code blocks (see →4.3.3 *More Is More Complex* (MIMC))
- For slightly unrelated but rather small pieces of functionality use private methods instead of an additional class.
- Avoid general solutions needing parameterization. A specific solution will suffice.
- ...

Caveats

See section contrary principles.

Origin

The principle was coined by the American engineer Kelly Johnson referring to the requirement that a military aircraft should be repairable with a limited set of tools under combat conditions [27].

The principle of striving for simple solutions sometimes is also called "(rule of) simplicity"[15] which was also prominently stated by Tony Hoare in his Turing Award lecture: "I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."[28]

Evidence

Accepted: This principle is widely known and accepted in practice. See for example Jargon File: KISS Principle [29].

Examined: While the preference for simple solutions can be considered trivially intuitive, there has been some work relating simplicity or rather complexity and certain quality attributes. But as there is no universally applicable complexity metric and not even a commonly agreed upon clear definition of simplicity, research is bound to examine certain aspects of KISS independently.

The following hypotheses can be stated:

- Simpler solutions are faster to implement.
- Simpler solutions yield less implementation faults (which reduces testing effort).
- Simpler solutions are easier to maintain, i. e. detecting and correcting defects is more effective and efficient.
- Simpler solutions yield more reliable software, i. e. less defects show up after releasing the software.

All these hypotheses can be examined with respect to different complexity metrics.

Hypothesis 1 can be regarded true by definition. If the solution cannot be implemented fast, it is not simple.

Though hypotheses 2 and 3 are not true by definition but they can be regarded intuitively clear. Nevertheless there is some research. In [30] a system was improved in two steps resulting in three variants of the same system. Several metrics show that the improvements reduced complexity. 36 programmers with varying experience conducted three different maintenance tasks and their performance was measured. The results indicate that the improvements also improved maintainability. Several other studies support the correlation between complexity and maintainability [31].

Furthermore software cost estimation techniques are partly based on complexity judgments [32]. So complexity—although this normally relates the complexity of the problem and not to the complexity of the solution—is a known cost factor which may be accounted to maintenance.

Lastly hypothesis 4 is likely to be false. Several studies relating complexity metrics and post-release reliability show that module size in lines of code predicts reliability at least as good as the McCabe metric (also called cyclomatic complexity)—see [17, p. 168ff.]. Assuming cyclomatic complexity correctly depicts the complexity of a module, reliability should not be the reason for applying KISS.

Relations to Other Principles

Generalizations

Specializations

- →4.3.3 *More Is More Complex* (MIMC): KISS states that one should strive for simplicity. MIMC makes this more concrete stating that more of anything (methods, classes, lines of code, ...) increases complexity.

Contrary Principles Note that many principles are contrary to KISS. This means that it is worthwhile to consider KISS when considering one of those. Nevertheless this does not mean that this is true the other way around. When considering KISS, one wouldn't want to consider all principles that have complexity as a disadvantage. So here are those needing consideration:

- →4.3.5 *Generalization Principle* (GP): This is the directly converse principle. A solution that is generally applicable typically is not simple anymore.
- →4.3.1 *Murphy's Law* (ML): The ultimate reason behind KISS is to increase maintainability and reduce the introduction of defects. But following KISS blindly by always using the simplest solution may also lead to *reduced* maintainability when Murphy's Law is not considered.
- →4.4.1 *Model Principle* (MP): There are often simpler ways to build a software system than to model and mirror the real world behavior, which frequently means having more objects and more complicated structures. Nevertheless it is advisable to do so anyway.

Complementary Principles

Examples

Example 1: Fuzzy Simplicity Simplicity is a blurry, partly subjective measure. Sometimes it is difficult to tell what is simpler. The following example shows that:

```
public String weekday1(int dayOfWeek)
{
    switch (dayOfWeek)
    {
        case 1: return "Monday";
        case 2: return "Tuesday";
        case 3: return "Wednesday";
        case 4: return "Thursday";
        case 5: return "Friday";
        case 6: return "Saturday";
        case 7: return "Sunday";
        default: throw new IllegalArgumentException("dayOfWeek must
            be in range 1..7");
    }
}

public String weekday2(int dayOfWeek)
{
    if ((dayOfWeek < 1) || (dayOfWeek > 7))
        throw new IllegalArgumentException("dayOfWeek must be in
            range 1..7");

    final String[] weekdays = {
        "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "
            Saturday", "Sunday"};

    return weekdays[dayOfWeek-1];
}
```

Both methods do exactly the same thing. They return a string representing the weekday. Just the implementation is different. Both versions may be seen as simpler than the other depending on the view taken. `weekday2` has less statements and less execution branches. Complexity metrics measuring these aspects (e. g. the cyclomatic complexity) will therefore prefer `weekday2`.

On the other hand `weekdays1` uses less language features (just switch, return, and exceptions whereas `weekdays2` needs if, arrays, arithmetic, return, and exceptions). Furthermore in `weekdays1` the relation between input and output can be seen directly and it's clear how it works by just seeing the method. But for understanding `weekday2` there are more details to think about. This is especially true for the range check at the beginning and the index computation in the **`return`** statement. Clearly this is not particularly difficult but these are aspects which are more difficult than in the other version of the method.

So it's not objectively clear which of the two implementations KISS prefers without saying which complexity metric to apply. But this ambiguity is not a problem since principles are not meant to be unambiguous and objective. Eventually a human developer has to decide which solution to implement and the principles only give guidelines.

4.3.3 More Is More Complex (MIMC)

Variants and Alternative Names

Principle Statement

More is more complex.

Description

Having more lines of code, methods, classes, packages, executables, libraries etc. always means also to have more complexity (which is bad). This means that given the complexity of the problem is fixed, a suitable compromise for the number of methods, classes, etc. has to be found. Reducing the number of statements per method typically results in the introduction of further methods. Reducing the number of methods per class can be achieved by dividing the class into several smaller classes, etc.

There is both: too large modules (i. e. under-modularization) and too small modules (i. e. over-modularization). Either there is too much complexity in a module (MIMC applied to one module) or there is too much complexity between the modules (MIMC applied to the number of modules).

Note that it is actually not the number of lines, methods, classes, etc. that is relevant but the effective number of items that have to be kept in mind for the purpose of understanding. So reducing the number of lines by placing several statements in one line does not help. Neither the introduction of an additional obvious private method will do any harm. MIMC is just a rule of thumb stating that the introduction of further modules (and the like) usually has a higher complexity as a drawback.

Rationale

The capabilities of the human mind are certainly limited. If it is necessary to keep a large amount of modules or lines of code in mind, it is difficult to understand. Furthermore if a module is large, it takes a long time to read (and thus to comprehend). And if there are many modules, looking for a particular module takes a long time. And the longer the searching process takes, the more one will have forgotten what has been read previously. This results in worse readability, understandability and this maintainability.

Strategies

- Avoid many modules
  - Merge several modules into one
  - Don't introduce a new module but put the functionality into another module
- Avoid big modules
  - Divide large modules in several smaller ones

Caveats

- Note that Miller's Law [33] stating that the human mind can remember "seven plus or minus two items" at at time is often cited in this context but it is doubtful if and to what extend it applies to design.
- Note that this principle is contrary to itself. Given a desired functionality a certain level of complexity in inevitable. This leads in the extremes either to a large amount of small classes or a large amount of code in a fewer class. The same applies on other levels like number and size of methods, etc. So there is always a tradeoff between MIMC and itself applied to different aspects of the software system.

See also section contrary principles.

Origin

The phrase "more is more complex" is new but can be regarded trivially intuitive to every developer. There is also some research concerning certain aspects of MIMC. See section evidence.

Evidence

Examined: There is some research relating module size to certain quality attributes like maintenance cost, error density, etc. Basili and Perricone studied maintenance data of Fortran programs for aerospace applications [34]. They found that the smaller modules had a higher error density than the larger ones. At first this seems to contradict MIMC. But assuming there is a certain essential complexity of the problem, this complexity has to be implemented somehow. Either this leads to a few large modules or many smaller ones. In the latter case the complexity is in the relationships and interactions between the modules instead of the modules themselves. So too small modules result in more modules and more complex communication among them. Other studies seem to confirm this[31].

This phenomenon that the defect density is high for small modules but also rises for large modules is called the "Goldilocks Conjecture". As a result there is an optimal module size which is neither too small, nor too big. Several publications claim to have found this optimal module size [35]. Depending on the programming language used, these values typically are claimed to be a few hundred lines of code. Note that most of these studies are in the context of procedural programming.

This sounds intuitive but the Goldilocks Conjecture is disputed. Some point out that the negative correlation between defect density and size is just a mathematical artifact [36][35] and that there are also other methodological problems with these studies [37]. There is also data which is not explainable by defect models based on the Goldilocks Conjecture [37].

The relationship between module size and defect proneness is complex and not clear. Furthermore modularization is not only a task in terms of module size. The more interesting aspect is how to assign responsibilities to modules. So apart from module size there are

many other aspects influencing modularization (see especially →4.4.1 *Model Principle* (MP), →4.5.2 *Low Coupling* (LC), and →4.4.2 *High Cohesion* (HC)) which makes it hard to isolate the pure effect of size.

This is an important research question but as MIMC is just a qualitative rule of thumb (just as the other principles are). So the principle can be deemed helpful despite the Goldilocks Conjecture being disputed.

As a specific aspect of MIMC, complexity through deep inheritance relations is known to reduce effectiveness and efficiency of maintenance. There are controlled experiments showing this[38][39]. On the other hand these results are limited as there may be many factors which are neglected by the experiment. Most notably in these experiments maintenance tasks where carried out on systems with artificially constructed inheritance hierarchies. It is undisputed hat there are good ways and bad ways of using inheritance. And it is doubtful that there are several equally good solutions for the same problem only differing in the depth of inheritance. So there is some evidence but no "proof" that deep inheritance hampers maintenance.

Questioned: The Goldilocks Conjecture, which can be seen as an aspect of MIMC, is disputed. See above.

Relations to Other Principles

Generalizations

- →4.3.2 *Keep It Simple Stupid* (KISS): MIMC states that having more modules, etc. leads to more complexity. KISS on the other hand is about the avoidance of every form of complexity.

Specializations

Contrary Principles Note that many principles are contrary to MIMC as they favor the introduction of additional modules. This means that it is worthwhile to consider MIMC when considering one of those. Nevertheless this does not mean that this is true the other way around. When considering MIMC, one wouldn't want to consider all principles that have complexity as a disadvantage. So here are those needing consideration:

- More Is More Complex (MIMC): Changing a design to adhere to the MIMC principle may always lead to more complexity concerning another aspect of the system. For example reducing the amount of code in a large method is typically achieved by the introduction of further methods. So there is always a tradeoff between this principle and itself.
- →4.4.2 *High Cohesion* (HC): Not introducing further modules typically leads to a lower cohesion.

Complementary Principles

Examples

4.3.4 Don't Repeat Yourself (DRY)

Variants and Alternative Names

- Single Point of Truth (SPOT)
- Single Source of Truth (SSOT)

Principle Statement

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."[13]

Description

DRY not only states that code duplication shall be avoided. Rather DRY is a general rule that states that if there is duplication, there shall be some "single source of truth". Also when one piece of information has several representations (like an object structure corresponding to a database schema) DRY demands one and only one representation being the definitive one. The other representations have to be generated automatically. The "one and only" representation can be one of the used representations or alternatively a third one.

Rationale

If there are several representations of the same information (be it code or any other form of information), all of them have to be maintained separately while changing at the same time. There is the danger that at some point in time the different representations diverge which is a fault. But if there is a single source of truth, there is only one place where changes have to be applied. Then the representations cannot diverge.

Strategies

- Add a new invokable module (a function, a method, a class, etc.) instead of duplicating code
- Use code generation when information has to be represented in multiple forms
- Use polymorphism to avoid repeatedly enumerating a set of possible solutions in if or switch statements

Caveats

See section contrary principles.

Origin

Andrew Hunt and David Thomas: *The Pragmatic Programmer: From Journeyman to Master* [13]

Evidence

- Examined: There is extensive research on code cloning, its reasons, automatic detection of code clones, their evolution, etc. In [40] Chanchal Kumar Roy and James R. Cordy present a 115 page survey on the state of research as of 2007. Many unanswered questions remain and research is still ongoing. In 2009 Juergens et al. analyzed fife systems written in C#, Cobol and Java each between around 200 and 500 kLOC [41]. They identified clones and intentionally and unintentionally inconsistent changes to them. They then related the faulty clones to them and came to the expected conclusion that clones are changed inconsistently and that this results in faults. So at least the part of DRY about code duplication is supported by research findings.
- Accepted: It is generally agreed upon that code duplication is to be avoided. But the broader meaning of DRY which results in the heavy use of code generators is often not considered. On the other hand *The Pragmatic Programmer* is a well known book which makes DRY a well-known and accepted principle.

Relations to Other Principles

Generalizations

- →4.3.1 *Murphy's Law* (ML): Duplication is a typical example for error possibilities. In case of a change, all instances of a duplicated piece of information have to be changed accordingly. So there is always the possibility to forget to change one of the duplicates. DRY is the application of ML to duplication.

Specializations

Contrary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): Especially code generators can be very complex.

Complementary Principles

- →4.3.5 *Generalization Principle* (GP): A generalized solution avoids duplication.

Examples

4.3.5 Generalization Principle (GP)

Variants and Alternative Names

- Build Generality into Software [16]
- Abstractions Live Longer than Details [13]

Principle Statement

A generalized solution, that solves not only one but many problems, is better than a specific solution.

Description

There are various ways to make a solution more generally applicable. In the simplest form this can be done by introducing a method with appropriate parameters. Other possibilities are classes, parametric types, callbacks, hook methods, etc.

A general solution abstracts from the specific tasks and solves a superset of them. Parameterization of some kind is used to specify what has to be done in a given situation.

A module can be more general than another one. But there are two aspects of this: First of all there is functionality. If module A can do the same as module B plus something more then A is more general. The second aspect is the one of what has to be done in order to exploit the generality. An ideal case would be that nothing has to be done and the module just does more. Other possibilities are that a configuration file has to be changed, an attribute has to be set, an invocation parameter has to be adjusted, etc. The least general possibility would be a module which can be changed easily. This is still better than a rigid module but less general than modules which do not need such changes. This form of generality is often rather called "flexibility" [42].

Rationale

Specific solutions tend to be fragile. When requirements change, a specific solution might not fulfill them anymore. In contrast to that a more general solution is more stable so there will be less need to change it.

Moreover a generalized solution can be reused in a variety of other situations. A specific solution can only be reused when exactly the same requirements appear again. So a general solution is much more reusable.

Strategies

- Make modules configurable at runtime or deployment time by using configuration files.

- Use parameterizable modules (method parameters, object attributes, parametric types, etc.)
- Use constants
- Find suitable abstractions

Caveats

Making a module (typically a layer, a subsystem or an API) too general may lead to the module mirroring the functionality of the underlying platform without adding a benefit but only complexity.

Another problem is the turing tarpit [43]. This means that the module is so general that arbitrarily complex tasks can be performed but those of interest, meaning the rather simple tasks that occur over and over again, are also difficult to do. This is a violation of the →4.6.1 *Easy to Use and hard to Misuse* (EUHM) principle.

See also section contrary principles.

Origin

The term "generalization principle" is proposed here. Nevertheless the value of generalized solutions is well known at least since:

David Parnas: *Designing Software for Ease of Extension and Contraction* [42]

Evidence

- Proposed: GP is assumed to be intuitive to every developer. Despite this fact—or maybe because of that—neither scientific examinations of its validity nor widely known publications describing it could be found.

Relations to Other Principles

Generalizations

Specializations

Contrary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): A generalized solution is typically not simple anymore. This is the typical conflict between generality and simplicity.
- →4.6.1 *Easy to Use and Hard to Misuse* (EUHM): Too general solutions may lead to complicated usage of the module.
- →4.3.6 *Rule of Explicitness* (RoE): RoE often results in specific solutions. Generality often requires stating something implicitly.

Complementary Principles

- →4.3.4 *Don't Repeat Yourself* (DRY): A more general solution avoids duplication.
- →4.4.3 *Encapsulate the Concept that Varies* (ECV): Encapsulating a varying concept typically results in a more generally applicable solution. This is especially true when an abstract concept is encapsulated by introducing an `interface` or an abstract class.

Examples

4.3.6 Rule of Explicitness (RoE)

Variants and Alternative Names

- Explicit Is Better Than Implicit (EIBTI) [44]

Principle Statement

"Explicit is better than implicit." [44]

Description

Solutions often differ in the level of explicitness. A feature can be implemented explicitly or it can be a side-effect of the implementation of another feature or a more general functionality. The same applies to module communication. A module can invoke another module directly or there can be various forms of indirections like events or observers.

RoE states that explicit solutions are better than implicit ones. Indirection, side-effects, configuration files, implicit conversions, etc. should be avoided.

Rationale

If something is realized explicitly, it is easier to understand. Implicit solutions require the developer to have a deeper understanding of the module as it is necessary to "read between the lines". Implicit solutions also tend to be more complex. So explicit solutions are assumed to be less error-prone and easier to maintain.

Strategies

- Avoid indirection (but keep →4.5.2 *Low Coupling* (LC) in mind)
  - Avoid indirection though events/listeners/observers, etc. and use direct references instead.
  - Avoid indirecting middleware like messaging middleware in favor of direct communication. Explicit communication paths are easier to grasp and debug.
- Avoid configurability (but keep →4.3.5 *Generalization Principle* (GP) in mind)

  - Avoid using configuration files for specifying behavior. Instead implement varying behavior explicitly.
  - Avoid highly configurable modules. Instead implement varying behavior explicitly.

- Explicitly state which module to use
  - Avoid importing all classes of a given package/namespace and import the needed classes explicitly. In Java this means not to specify wildcard imports like `import package.*` and to avoid static imports. Similarly in Python this means not to use wildcard imports.
  - Avoid `with` statements in Delphi and other languages having constructs that let you invoke methods without explicitly stating the associated object.

- Explicitly name parameters
  - In Python and other languages that allow this, use named parameters.
  - Avoid long parameter lists and use objects with explicit attribute assignments instead.
  - Use parameter types that explicitly state what the input is. Rather use specific types for parameters like customers, articles, URLs, colors, money, etc. instead of using strings or integers for these values.

- Avoid implicit type conversions.
  - In C# do not to specify implicit cast operations
  - In C++ use the `explicit` keyword on single-parameter constructors
  - In PHP use the `===` operator instead of `==` where the type matters

Caveats

See section contrary principles.

Origin

- First without being explicitly stated RoE has been a central design principle of the programming language Python[45]. Python dates back to 1991.
- Later this philosophy was stated as part of the “Zen of Python”[44].
- The rule—although often not stated as such—is also known outside the python community[46].
- Extend and origin beyond that remains unclear.
- The name “rule of explicitness” is newly introduced here.

Evidence

- Accepted: Explained by Martin Fowler in [46] and in virtually every Python book.

Relations to Other Principles

Generalizations

Specializations

Contrary Principles

- →4.3.3 *More Is More Complex* (MIMC): Stating something explicitly requires more code.
- →4.3.5 *Generalization Principle* (GP): RoE often results in specific solutions. Generality often requires stating something implicitly.
- →4.5.2 *Low Coupling* (LC): Direct communication typically has the disadvantage of a higher coupling. Indirection reduces coupling but creates implicit/indirect communication paths.

Complementary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): Explicit solutions are often also simpler.
- →4.3.1 *Murphy's Law* (ML):The typical reason for RoE is to avoid unnecessarily complicated solutions and possibilities for defects. Don't loose that goal out of sight.
- →4.4.1 *Model Principle* (MP): RoE states that "primitive obsession" shall be avoided. This means that primitive types should not be used as parameters if there is the possibility to specify a more specific object type. MP makes this even clearer: In object-orientation objects instead of plain integers or strings are used.

Examples

4.4 Modularization Principles

4.4.1 Model Principle (MP)

Variants and Alternative Names

- Direct Mapping [14]
- Low Representational Gap (LRG) [12, p. 281]

Principle Statement

The object structure of the software should model and mirror those concepts and actions of the real world, that the software supports.

Description

The software should model and mirror the "real world". This first of all means, that the structure of the software—to some extend—models the structure of the problem. When the "real world action" that the software should support comprises certain entities like e. g. customers, products, and orders, then there should be one object for each customer, product and order. Furthermore there should be one class for each concept. And if there is a certain relationship between customers, orders, and products, there should also be an association between the corresponding classes and references between the objects. So the object structure models the structure of the real world concepts.

Real world actions are then *mirrored* in the software system. This means that each action in the real world triggers a corresponding action in the model world which ensures that the model stays consistent with the real world. So the software is a kind of a simulation of what actually happens. If customer orders some product, the software reacts by creating an order object, which is connected to the customer and the product objects corresponding to the customer and the product involved in the real world action. This may be done by a method called `Customer.orderProducts()`.

Rationale

When the structures in the software roughly correspond to the structures of the problem domain, a developer doesn't have to learn both of them. Knowing the problem domain is inevitably necessary. Any further structure of the software has to be learned and understood in addition. So creating a direct mapping between them, makes understanding the software easier, which improves maintainability. In such a system for most functionality there is a "natural", i. e. an intuitively clear place to implement it. This makes structuring the software easier and helps finding the implementation for a given functionality.

Strategies

- Create a class for each relevant real-world concept ("natural classes")
- Create methods corresponding to real-world actions
- Map additionally necessary behavior to natural classes instead of creating artificial classes
- For artificial behavior that cannot be mapped to a natural class at least create a metaphor or an artificial model (like for example a state machine)

Caveats

This principle may lead to the problem of modeling the real world in too great detail. This complicates the design without giving any further benefits. Especially a wrong understanding of inheritance may lead to "taxomania", where an inheritance relation and the respective classes are only introduced because there seems to be such a taxonomy in the "real world"[14]. But inheritance should only be used on purpose, not just because it is possible. A special form of this problem is called "vapor classes", which are useless abstractions which are never used[10].

See also section contrary principles.

Origin

The root of this principle is the very beginning of object-orientation itself. The idea behind Simula, the first object-oriented programming language, was to view program executions as simulations. Kristen Nygaard, one of the creators of Simula defines object orientation as follows: "Object-oriented programming. A program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world."—see for example [47].

Although this view is disputed as a definition for object-oriented programming, it became the key idea of object-oriented analysis. In [48, p. 191] Grady Booch clearly states that objects "directly reflect our model of reality".

Evidence

- Accepted: Virtually every introduction to object-oriented analysis roughly explains this but mostly without stating it as a principle. Bertrand Meyer explains this principle in his book *Object-Oriented Software Construction* [14]
- Questioned: The value of this principle is disputed. It is questioned whether objects in the OOP sense nicely map to real-world objects[49][50]. Furthermore there is the typical object-relational impedance mismatch and the observation that business rules are sometimes cross-cutting [51]. There also is not one single obvious model for the "real world". A model is subjective to the one creating the model. So it is not enough to model the "real world" but it is important to think about how to model it [14, p. 694].

Relations to Other Principles

Generalizations

Specializations

Contrary Principles

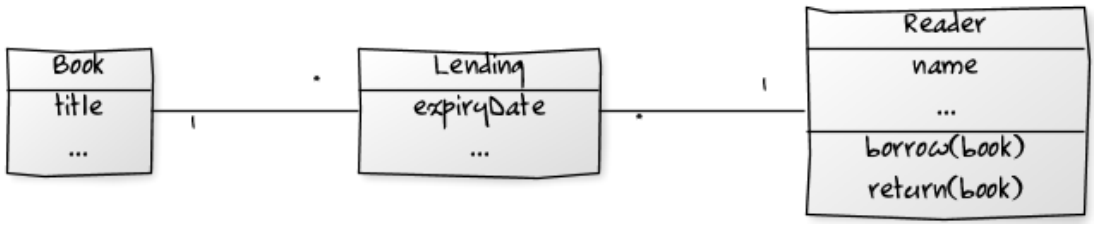
- →4.4.3 *Encapsulate the Concept that Varies* (ECV): Sometimes there are "concepts that vary" which are not directly related to a real-world concept. So ECV demands having an artificial class.
- →4.3.2 *Keep It Simple Stupid* (KISS): There are often simpler ways to build a software system than to model and mirror the real world behavior, which frequently means having more objects and more complicated structures.
- →4.4.2 *High Cohesion* (HC): MP sometimes creates classes with suboptimal cohesion.

Complementary Principles

- →4.5.1 *Tell, don't Ask/Information Expert* (TdA/IE): TdA/IE tells how to distribute functionality among the natural classes which are created according to the Model Principle.
- →4.5.2 *Low Coupling* (LC): When designing a model for a software, it has to be borne in mind that structures with low coupling are desirable.

Examples

Example 1: Object Structure (Library) In a software system for a library, there will be a classes like `Book`, `Reader`, and `Lending`. A reader has a name, a book has a title and the reader must return the book after some date of expiry. So the corresponding classes will have attributes describing these properties. The reader may borrow and return a book, so the `Reader` class will have methods `borrow()` and `return()`. Classes, attributes, and methods are directly inferred from the problem domain.



Example 2: Swing GUI frameworks like Java Swing typically have classes corresponding to the types of controls that can be used to build graphical user interfaces. So Swing for example has classes like `JButton`, `JCheckBox`, and `JTextField`.

Furthermore buttons, check boxes, text fields, and the like are also models of concepts in the real world. Buttons are typical controls of machines and check boxes and text fields are parts of a typical (paper-based) form. So the class `JCheckBox` is a model for a check box on the screen which itself is a model for a check box on a paper-based form.

Example 3: Dependencies MP also tells which modules may depend on which others. Suppose there is a software comprising a parser for mathematical functions. Obviously there will be classes `Parser` and `Function`. MP tells that dependencies between these classes shall be according to the model. Logically a parser parses a string and creates `Function` objects. It is impossible to think about the `Parser` without `Functions`. So `Parser` may naturally depend on `Function`.

On the other hand our intuitive model of parsers and functions tells us that a `Function` does not need a `Parser` to be a meaningful entity. One can easily think of `Functions` created by using builder methods instead of a parser. And even if that wasn't true and there would only be the possibility to create functions by using parsers, a `Function` object logically can work without knowing that there are parsers which have created it. In an imaginary hierarchy of modules Parser would be a module on a higher scale than `Function`. So MP forbids that `Function` depends on `Parser`.

4.4.2 High Cohesion (HC)

Variants and Alternative Names

Principle Statement

Cohesion in a module should be high.

Description

The cohesion of a module is a measure for how well the internal parts of a module (e. g. the methods and attributes of a class) belong together. Having a high cohesion means, that a module should only comprise responsibilities which belong together.

Rationale

Not adhering to this principle, i. e. having a low cohesion, means that one module has several unrelated or only loosely related responsibilities. A change in the requirements for one of these may thus also affect the others which would not be the case in a highly cohesive module.

Strategies

- Divide one large module into several smaller but more cohesive ones

Caveats

See section contrary principles.

Origin

W. P. Stevens, G J. Myers, L. L. Constantine: *Structured design* [52]

Evidence

- Examined There are metrics that try to measure cohesion. There are studies relating these cohesion measures to the number of errors found during testing [17]. This correlation is evident. The limitation of these studies is that these cohesion metrics cannot represent the cohesion notion completely.
- Accepted The concept of high cohesion is widely known and described in several well-known books for example in Craig Larman's *Applying UML and Patterns* [12].

Relations to Other Principles

Generalizations

Specializations

Contrary Principles

- →4.3.3 *More Is More Complex* (MIMC): Making a module highly cohesive often results in additional modules. Sometimes it is simpler to assign a minor unrelated responsibility to a module, which lowers the cohesion.
- →4.4.1 *Model Principle* (MP): Adhering to HC sometimes means to split up a class into several smaller ones which might correspond to the model less well.
- →4.5.2 *Low Coupling* (LC): A system consisting of one single module has a very low coupling as there are no dependencies on other modules. But such a system also has low cohesion. The other extreme, very many highly cohesive modules, naturally has a higher coupling between the modules. So here a compromise has to be found.

Complementary Principles

- →4.5.1 *Tell don't Ask/Information Expert* (TdA/IE): IE may help finding solutions with high cohesion. On the other hand it may also be disadvantageous in some cases (see →4.5.1 *Tell don't Ask/Information Expert* (TdA/IE), caveats section).
- →4.4.3 *Encapsulate the Concept that Varies* (ECV): Adhering to HC often results in modules to be split up into several more cohesive ones. ECV gives further advice on how to do that.

Examples

4.4.3 Encapsulate the Concept that Varies (ECV)

Variants and Alternative Names

Principle Statement

Encapsulate the Concept that Varies means a design is better when those parts that vary are encapsulated in a separate module.

Description

This principle has two aspects. The first one is about making changes local. Everything which is supposed to change in the future should be encapsulated in a single module. This means cross-cutting concerns are avoided as much as possible. This is not completely possible but in many cases it is.

The second aspect is about introducing abstractions. Sometime the varying concept is one which varies at runtime rather than by maintenance. So at runtime it is decided upon a certain variation or there can be even several variations at the same time. In this case there has to be an abstract base class or an `interface` which encapsulates the varying concept. Several concrete descendant classes then specify the concrete variation.

The difference between the two aspects is whether the varying concept is one that changes over time during maintenance or one that may change at runtime. Nevertheless the advice is the same: encapsulate the concept that varies.

Rationale

There are two reasons for this principle. The first reason is locality. When a varying concept is properly encapsulated in a single module, only this module is affected in case of a change. This reduces maintenance effort and ripple effects.

The second reason comes to play when the varying concept is implemented as an abstract class or `interface`. In this case a variation can be introduced without changing existing and tested code. This reduces testing effort (as already tested code does not need to be retested as it is not changed) as well as ripple effects (as the enhancement is done simply by adding a class. Note that for this rationale to work, the →4.7.3 *Liskov Substitution Principle* (LSP) also has to be adhered to.

Strategies

- Introduce a separate module for the concept that may change in the future. In that way the future change will only affect that particular module. If the varying concept is properly encapsulated, only this module will have to change.
- Introduce an `interface` encapsulating the varying concept. The `interface` may be implemented differently by several classes and code that only relies on the `interface`

can handle any class implementing the interface. In case of another variation, just another class has to be introduced and this class has to implement the `interface`. If the abstraction is done properly, no module has to change.

- Introduce an abstract base class encapsulating the varying concept. This is basically the same as introducing an `interface`. But here, implementation can also be inherited. So common parts can remain in the abstract base class whereas only the actual variations are defined in the subclasses. By means of method overriding, the implementation of the base class methods can be changed without touching the base class directly.
- Use design patterns. Several design patterns use the above techniques to encapsulate varying concepts. For example:
  - Abstract Factory: A family of objects changes.
  - Factory Method: The exact type of an object to create changes.
  - Adapter: The interface of a module changes.
  - Bridge: A concept varies in more than one aspect.
  - Decorator: The behavior of certain methods may need to be enhanced.
  - Iterator: The traversal algorithm of a structure changes. Or the structure itself changes resulting in the need for a different traversal algorithm.
  - Observer: The objects interested in a certain event may change.
  - State: The behavior in a certain state or the state machine (states and transitions) of a certain module changes.
  - Strategy: An algorithm changes.
  - Template Method: The concrete steps in an algorithms change.
  - Visitor: New operations have to be added to a given more or less static inheritance structure of classes.
  - ...

Caveats

See section contrary principles.

Origin

The principle is stated, explained and used in the "GoF book": Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* [3, p. 29] But the idea of ECV is actually much older. It was first presented in David Parnas: *On the Criteria To Be Used in Decomposing Systems into Modules* [53]

Evidence

- Accepted: This principle was popularly described in the GoF book and can thus be regarded accepted.
- Examined: Many of the patterns in the GoF book are precisely about encapsulating varying concepts. See strategies section.

Relations to Other Principles

Generalizations

Specializations

Contrary Principles

- →4.3.3 *More Is More Complex* (MIMC): ECV demands adding a new class for a new varying concept.
- →4.4.1 *Model Principle* (MP): ECV sometimes results in classes which do not correspond top a real-world concept in the sense of MP. A "concept that varies" can also be a technical concept.

Complementary Principles

- →4.5.2 *Low Coupling* (LC): ECV results in the creation of a new module. When introducing such a new module, LC has to be adhered to.
- →4.7.3 *Liskov Substitution Principle* (LSP): ECV may result in the introduction of an abstract base class. Here it is important to get the abstraction right. Otherwise LSP may be violated.
- →4.3.5 *Generalization Principle* (GP): Encapsulating a varying concept typically results in a more generally applicable solution. This is especially true when an abstract concept is encapsulated by introducing an `interface` or an abstract class.
- →4.5.3 *Dependency Inversion Principle* (DIP): ECV may result in the introduction of an abstract base class. Here DIP demands that other classes should only depend on this new abstract base class and not on the concrete subclasses.
- →4.7.1 *Information Hiding/Encapsulation* (IH/E): ECV tells that varying concepts should be encapsulated. IH/E then tells how encapsulation is done.

Examples

4.5 Module Communication Principles

4.5.1 Tell Don't Ask/Information Expert (TdA/IE)

Variants and Alternative Names

- Information Expert or Expert in [12]
- Do It Myself in [54]
- Tell, don't Ask in [55]

Principle Statement

- Assign a responsibility to that module which has the largest subset of the required information.
- Don't ask an object for information, make computations, and set values on the object later. Tell the object what it should do.

Description

Each module has a set of responsibilities. Subsystems have specific tasks, packages group several related classes, classes have methods and attributes, and so on. So there is a kind of mapping between modules and responsibilities. This mapping is good when the information which is necessary to fulfill the given task is present in the given module so there is no need to acquire all this information.

Rationale

When this principle is not adhered to, then a module has a responsibility for which it is lacking some information. So in order to fulfill the task the module has to first acquire the needed information by invoking other modules. This increases the dependencies between the modules (which may lead to ripple effects).

Furthermore not adhering to TdA is a sign of not thinking in an object-oriented way. In procedural programming there is a clear separation between data structures and procedures. Applying this thinking to object-oriented languages leads to objects which have no functionality besides having trivial getter and setter methods making it a dump data structure instead of an object carrying real behavior.

Strategies

- Assign a responsibility to the class that has the largest subset of the needed information.

- Mirror functionality of composed objects to the interface of the class instead of having a getter-method returning the composed object
- Have the objects operate on their own data using appropriate methods. Avoid getters and setters.

Caveats

Sometimes assigning responsibilities using IE results in bad solutions (high coupling, low cohesion). This is because IE just focuses on the availability of data. So for example IE would demand domain objects saving themselves to the database. This is bad since it couples the domain objects to the database interface (JDBC, SQL, etc.) and lowers cohesion by adding unrelated responsibilities to the classes. Here it is better to give the task of persisting the domain objects to a separate class [12, p. 298].

See also section contrary principles.

Origin

Craig Larman: *Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development* [12]

Evidence

Accepted: This principle is prominently described in Craig Larman's book *Applying UML and Patterns*[12].

Relations to Other Principles

Generalizations

Specializations

Contrary Principles

- →4.3.3 *More Is More Complex* (MIMC): Adhering to TdA/IE sometimes results in adding further methods.

Complementary Principles

- →4.5.2 *Low Coupling* (LC) Adhering to IE typically leads to low coupling as there is less need to communicate with other modules to get the necessary information. But in some cases IE also increases coupling (see section caveats).
- →4.4.2 *High Cohesion* (HC) Adhering to IE typically leads to high cohesion as responsibilities which belong together typically operate on the same data. But in some cases IE also lowers cohesion (see section caveats).

- →4.4.1 *Model Principle* (MP): TdA/IE tells how to distribute functionality among the natural classes which are created according to the Model Principle.
- →4.7.1 *Information Hiding/Encapsulation* (IH/E): Assigning responsibilities to objects using Information Expert may accidentally break encapsulation. It typically does not but it has to be considered. Furthermore TdA is about not having getter methods returning constituent parts of a module. Encapsulation can be another reason for that.
- →4.7.4 *Principle of Separate Understandability* (PSU): TdA/IE is about responsibility assignment. Another aspect of this task is treated by PSU.

Examples

4.5.2 Low Coupling (LC)

Variants and Alternative Names

- Loose Coupling

Principle Statement

Coupling between modules should be low.

Description

A module should not interact with too many other modules. Furthermore if a module A interacts with another module B, this interaction should be loose, which means that A should not make too many assumptions about B.

Coupling is a measure of dependency between modules. The more dependencies there are, the stronger the dependencies are, and the more assumptions are made upon other modules, the higher is the coupling.

There are different forms of couplings which can be rated according to their strength [56]:

No coupling The modules do not know each other.

Call coupling A module calls another one.

Data coupling A module calls another one passing parameters to it.

Stamp coupling A module calls another one passing complex parameters to it.

Control coupling A module influences the control flow of another module.

External coupling The modules communicate using a simple global variable.

Common coupling The modules communicate using a common global data structure.

Content coupling A modules depends on the inner working of another module. This is the strongest form of coupling.

The forms ranging from no coupling to stanp coupling can be considered "good" couplings. The others are rather strong.

There are also some additional forms of undesirable couplings:

Tramp coupling A module is only coupled to a data structure because some other module needs the data. The module gets the data and passes it to the other module without touching the "tramp data" [57].

Logical coupling A module makes some assumptions about another module without referencing it. For example a module A only sorts a list because some other module B which A technically does not know about needs it sorted.

Rationale

If a module A interacts with a module B, there is a certain dependency between these modules. When for example A uses a certain functionality of B, then A depends on B. A makes the assumption that B provides a certain service, and moreover it makes assumptions on how this service can be used (by which mechanism, which parameters, etc.). If one of these assumptions is not true anymore because B has changed for some reason, A also has to change. So the fewer dependencies there are, the less likely it is that A stops working and has to be changed.

Furthermore A makes many and detailed assumptions about B, there is also a high probability that A has to change despite only relying one one other module. This is because in such a case A also needs to change when only a certain detail of B changes.

But if coupling is low, there are only few assumptions between the modules which can be violated. This reduces the chance of ripple effects.

Strategies

- Indirection: Don't access the other module directly but have another module do that.
- Use the observer pattern [3]
- Use lower forms of coupling
- Merge modules: when there is only one module, then there is no communication and thus no coupling
- Hide information: Information which is hidden cannot be depended upon.

Caveats

Coupling can be reduced by several technical measures (see section strategies). But while these measures reduce the coupling technically, they do not necessarily reduce the logical coupling. In such a case two modules A and B may seem decoupled, but ripple effects may occur anyway because they make assumptions about each other. In such a case it is better to make the coupling explicit by not applying a decoupling strategy. It may also be possible to find a better suitable strategy or a better way of applying the strategy to also get rid of the logical coupling.

Furthermore note that coupling to a stable module is often no problem. The problematic cases are couplings to unstable modules. This means that applying decoupling strategies is beneficial when a coupling to an unstable module is reduced. But it may not be beneficial in the other cases.

See also section contrary principles.

Origin

W. P. Stevens, G J. Myers, L. L. Constantine: *Structured design* [52]

Evidence

- Examined: There are metrics that try to measure coupling. There are studies relating these coupling measures to the number of errors found during testing [17]. This correlation is evident. The limitation of these studies is that these coupling metrics cannot represent the coupling notion completely.
- Accepted: The concept of low coupling is widely known and described in several well-known books for example in Craig Larman's *Applying UML and Patterns*

Relations to Other Principles

Generalizations

Specializations

- →4.5.3 *Dependency Inversion Principle* (DIP): LC aims at reducing the dependencies to other modules. One way to do so is to only depend on abstractions. DIP is about this aspect.

Contrary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): Reducing the coupling often involves the use of complicated interaction patterns, indirections, etc.

- →4.4.2 *High Cohesion* (HC): A system consisting of one single module has a very low coupling as there are no dependencies on other modules. But such a system also has low cohesion. The other extreme, very many highly cohesive modules, naturally has a higher coupling between the modules. So here a compromise has to be found.
- →4.3.6 *Rule of Explicitness* (RoE): Direct communication typically has the disadvantage of a higher coupling. Indirection reduces coupling but creates implicit/indirect communication paths.

Complementary Principles

- →4.5.1 *Tell, don't Ask/Information Expert* (TdA/IE): IE may help to reduce coupling. Although there are also contrary cases (see →4.5.1 *Tell, don't Ask/Information Expert* (TdA/IE), section caveats).
- →4.4.1 *Model Principle* (MP): LC aims at reducing the dependencies to other modules. So a module shall depend on only a few others. MP now tells which dependencies are allowed and which aren't.
- →4.7.1 *Information Hiding/Encapsulation* (IH/E): Higher forms of couplings (especially content couplings) break encapsulation.

Examples

4.5.3 Dependency Inversion Principle (DIP)

Variants and Alternative Names

Principle Statement

"Depend on abstractions" [9, p. 129].

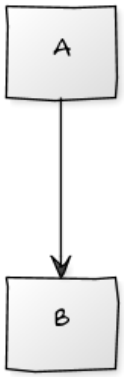
Description

A simplified description of DIP is that a variable declaration should always have the (static) type of an abstract class or `interface`. By doing so a module depends only on this abstraction. The concrete subclass realizing the details is referenced only once, namely when it is instantiated.

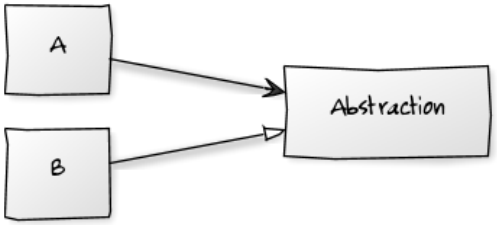
The more elaborate definition by Robert C. Martin reads as follows:

"a. High-level modules should not depend on low-level modules. Both should depend on abstractions.
b. Abstractions should not depend on details. Details should depend on abstractions."[9]

Following this rule leads to "inverted" dependencies compared to classical procedural approaches. The following diagram shows the classical approach. A high-level module *A* uses a low-level module *B*.



When applying DIP, both modules depend on the abstraction (note that in UML diagrams all arrows point into the direction of the dependency):



B is not depended upon anymore but it depends on another module. This is the inverted dependency.

Rationale

When DIP is not applied, only the low-level modules can be reused independently. The higher-level modules depend on the others, so trying to reuse them makes it necessary to either also reuse the lower-level modules or to change the higher-level module. The former is often not wanted because reuse is often done in another context where the lower-level modules do not fit. And the latter is error-prone and requires additional work as it requires changes to already working modules.

Strategies

- Use events, the observer pattern, etc. to remove dependencies
- Have an `interface` type for every class
- Declare only `interface` types so that an object variable generally has an `interface` as static type and a concrete class as dynamic type
- Do not derive classes from concrete ones (i. e. non-abstract classes)
- Do not override already implemented methods in subclasses

Caveats

It is normally not helpful to apply DIP to value objects like classes for amounts of money, date and time, or air pressure.

See also section contrary principles.

Origin

Robert C. Martin: *Object Oriented Design Quality Metrics: An Analysis of Dependencies* [58]

Evidence

Accepted: DIP is part of the well-known SOLID principle collection [9].

Relations to Other Principles

Generalizations

- →4.5.2 *Low Coupling* (LC): LC aims at reducing the dependencies to other modules. One way to do so is to only depend on abstractions. DIP is about this aspect.

Specializations

Contrary Principles

- →4.3.3 *More Is More Complex* (MIMC): DIP demands introducing abstractions, especially abstract classes or `interfaces`.

Complementary Principles

- →4.4.1 *Model Principle* (MP): DIP demands having abstractions. MP tells how these abstractions can look like.

Examples

Example 1: Furnace An example for a high-level module is a regulator module of a furnace. The classical approach would result in the regulator depending on a thermometer and a heater. in such a case it would not be possible to reuse the regulator module for regulating the fluid level of a reservoir or the speed of a car. A DIP-compliant solution would result in the regulator just depending on a sensor module and a actuator module and thermometer and header implementing these `interfaces`. By doing so thermometer, heater, and regulator can be reused independently.

This example is taken from [9] and slightly modified.

4.6 Interface Design Principles

4.6.1 Easy To Use And Hard To Misuse (EUHM)

Variants and Alternative Names

Principle Statement

A module shall be easy to use and hard to misuse.

Description

A module is easy to use when the obvious way of using it is correct, when following established conventions means to use it correctly and the identifier of the module hints the correct usage. A module is hard to misuse, if misusing it requires more work than correct usage and when the compiler signals wrong usages.

Rationale

This principle is common wisdom among API designers. APIs are used by many people and mainly by those who have not implemented the API. Few people read the documentation and will just try to use an API the obvious way. If the obvious way of using it is correct, if the API is easy to use, development will be more efficient. Furthermore if an API is hard to misuse, fewer mistakes will be made and which decreases the need for debugging and improves code quality.

The same reasoning holds for every module, not just APIs. There are always other developers who will use a module. Either team members or successors. Moreover after a while one will not remember the details of a module anymore, the difference between other people and oneself will vanish. This means EUHM not only holds for APIs and not only for interfaces provided to the other team members but it holds for every single module.

Strategies

This is a very general principle so there is a large variety of possible strategies to adhere more to this principle largely depending on the given design problem:

- Make use of static typing, so the compiler will report faults
- Make the interface simple, so there will be fewer usage defects (see →4.3.2 *Keep It Simple Stupid* (KISS))
- Use the same mechanisms wherever reasonably possible (see →4.6.3 *Uniformity Principle* (UP))
- Use consistent naming and models throughout the design (see →4.6.3 *Uniformity Principle* (UP) and →4.4.1 *Model Principle* (MP))

- Avoid Preconditions (see →4.7.2 *Invariant Avoidance Principle* (IAP))
- ...

Caveats

See section contrary principles.

Origin

The precise origin of the principle is unknown.

Evidence

Proposed

Relations to Other Principles

Generalizations

- →4.3.1 *Murphy's Law* (ML): Because of ML an interface should be crafted so it is easy to use and hard to misuse. EUHM is the application of ML to interfaces.

Specializations

- →4.6.2 *Principle of Least Surprise* (PLS): A module is easy to use if there is no surprise in how it works.

Contrary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): Both principles, KISS and EUHM, are about simplicity. But while EUHM is about the simplicity of an interface, KISS is rather concerned with simplicity of the implementation. KISS is contrary in those cases where the solution which is easier to implement is not so easy to use or imposes further possibilities for misuse.

Complementary Principles

- →4.4.1 *Model Principle* (MP): An interface that is crafted according to the model is easier to use than one that is not.
- →4.7.2 *Invariant Avoidance Principle* (IAP): One reason for a possible misuse of a module is an invariant. For example there might be a method which takes a list and an index where the index has to be within the bounds of the list. Each of these invariants imposes further possibilities for misuse of the module. So it is better to avoid them.
- →4.7.1 *Information Hiding/Encapsulation* (IH/E): A module should be properly encapsulated in order to make it easy to use and hard to misuse.

Examples

4.6.2 Principle Of Least Surprise (PLS)

Variants and Alternative Names

- Principle of Least Astonishment (PLA)
- May also be referred to as “rule” or “law” instead of “principle”
- Acronyms sometimes include the “o” for “of”: PoLA, PoLS

Principle Statement

“In interface design, always do the least surprising thing.”[15]

Description

Never surprise the user. An interface should behave exactly as the user thinks it behaves. What surprises the user depends on the kind of interface (user interface, module interface) and the type of user (end user, fellow programmer, maintainer). The central idea of PLS is to think about how the user would want to use the interface.

Rationale

Surprises are always a potential source for frustration. A user wants to be in control of the system. If the system does not behave as intended, the user gets disappointed and has to determine how to get the system do what it should do. On the other hand a system that behaves according to the users wishes is pleasant to use.

Secondly when everything works as expected, the user will make fewer mistakes. In case of a user interface this means that the user is more effective and in case of a module interface the software will have fewer defects.

Strategies

- Separate methods that change an object (commands) from methods asking the object a question (queries)
  - This especially means that a query method should not alter the observable object state
- Name all modules in a way that clearly communicates what the module is and does
  - Names of classes shall be nouns representing a specific (real-world) concept (see →4.4.1 *Model Principle* (MP))
  - Names of `interfaces` shall be adjectives describing a specific property. This typically results in names ending with -able

  - Names of command methods shall be verbs (in imperative form)
  - Names of query methods shall start with get- or is-
  - Names of mathematical functions or the like shall be named by the respective concept (like `sqrt`)
  - ...

- Avoid "clever" solutions which are hard to grasp in favor of simple, dumb ones (see →4.3.2 *Keep It Simple Stupid* (KISS))
  - When in doubt, use brute force [15]
  - Tend to use the first solution that comes in mind

Caveats

See section contrary principles.

Origin

The precise origin is unknown. Probably it's *The Tao Of Programming* by Geoffrey James [59].

Evidence

Accepted: PLA is widely known and also treated in Eric S. Raymond's *The Art of Unix Programming*

Relations to Other Principles

Generalizations

- →4.6.1 *Easy to Use and Hard to Misuse* (EUHM): A module is easy to use if there is no surprise in how it works.

Specializations

Contrary Principles

Complementary Principles

- →4.4.1 *Model Principle* (MP): PLS is mainly about how module identifier and module behavior relate to each other. MP tells that modules named according to the model are least surprising.
- →4.6.3 *Uniformity Principle* (UP): When applying PLS, UP should also be considered for naming modules.

Examples

4.6.3 Uniformity Principle (UP)

Variants and Alternative Names

Principle Statement

Solve similar problems in the same way.

Description

Software design comprises many similar tasks. There are plenty of design decisions that are similar to ones taken before. UP tells that a design is good when similar design problems are solved the same way. UP can be applied to a large variety of problems: naming identifiers, ordering parameters, deciding upon framework or library usage, etc.

Striving for consistency and always using the same solutions also means that it can be a good idea to apply a "bad" or less-well suited solution for the sake of consistency. If for example a bad naming scheme is used throughout the whole project, it is advisable not to break it as an inconsistency in the naming scheme would be worse than applying the bad naming scheme everywhere.

Rationale

Following UP reduces the number of different solutions. There are fewer concepts to learn, fewer problems to solve and fewer kinds of defects that can occur. So the developers, whether the original ones or the maintainers, have an easier task in creating, understanding, and maintaining the software. By reducing variety in the design, the software becomes easier (see →4.3.2 *Keep It Simple Stupid* (KISS)).

Strategies

- Use the same naming scheme everywhere
- Use the same techniques, mechanisms, libraries, and frameworks everywhere
- In similar methods use the same order of parameters

Caveats

UP demands solving similar problems in the *same way* and not just in a similar way. This is crucial as subtle differences can be dangerous. These small differences are created easily. Sometimes it is impossible to do two things exactly the same way. And also over time two modules may slowly diverge. So it is sometimes better to have two modules work completely differently than to allow for these subtle differences as they easily lead to misconceptions and mistakes (see →4.3.1 *Murphy's Law* (ML)).

See also section contrary principles.

Origin

This principle is newly proposes here. Nevertheless the idea is not new and should be pretty intuitive to every developer.

Evidence

- Proposed

Relations to Other Principles

Generalizations

- →4.3.1 *Murphy's Law* (ML): A typical source of mistakes are differences. If similar things work similarly, they are more understandable. But if there are subtle differences in how things work, it is likely that someone will make the mistake to mix this up.

Specializations

Contrary Principles Note that UP can be contrary to virtually every other principle as it demands neglecting other principles in favor of uniformity.

- →4.3.2 *Keep It Simple Stupid* (KISS): Although UP normally reduces complexity, sometimes UP demands more complex solutions because they are already applied elsewhere and for the sake of uniformity shall also be applied in simpler contexts where they would not be necessary.
- →4.4.1 *Model Principle* (MP): UP may demand adhering to a certain naming scheme, which may not be best with respect to MP. See example 1: naming schemes.

Complementary Principles

- →4.6.2 *Principle of Least Surprise* (PLS): When applying UP, PLS should also be considered for naming modules. See example 1: naming schemes.

Examples

Example 1: Naming Schemes A typical example of the application of UP is the naming of method identifiers for common container classes like stacks or queues. This also shows that there are several ways to apply this principle.

Stacks typically have the methods `push`, `pop` and `peek` (sometimes also called `top`). `push` puts an item onto the stack, `pop` removes the top most item and `peek` retrieves the value of the top most item without removing it from the stack. This is how the common stack model describes this data structure (see →4.4.1 *Model Principle* (MP)). Applying UP to this naming decision means that the methods should be named precisely as they are named

everywhere else also. So a developer knowing the model or other implementations of the model will immediately know how to use this module as well. In this case MP and UP demand the same thing. →4.6.2 *Principle of Least Surprise* (PLS) is satisfied here as well as a developer knowing stacks will expect exactly that.

Queues on the other hand typically have the methods `enqueue`, `dequeue`, and `peek` (or `front`/`first` or the like). MP would demand naming the operations of a `Queue` module exactly that way. But there are several ways Up can be applied here. The one way is to apply the principle just like above. Resulting in methods `enqueue` and `dequeue`. This is how it is done in .NET[60]. The other way is to consider the method identifiers of the `Stack` module. A possible application of UP could be to demand naming the queue methods just like the stack methods, meaning also `push`, `pop` and `peek`. This is the naming scheme which was chosen in the Delphi RTL [61]. Here MP and UP are contrary. A further downside of this approach is that `pop` and `push` methods might be surprising for a queue class. So PLS would oppose this solution.

A third possibility is to find a common abstraction and to apply a very general naming scheme to all descendant classes (stack classes, queue classes and others). This is the way it is done in Eiffel[14, p. 127]. Here there the method names are `put`, `remove` and `item` regardless of the concrete data structure. This is contrary to MP but creates a uniform naming scheme throughout the API. So there is less uniformity across APIs but stronger uniformity within the API. MP and UP are here contrary too. For PLS this means that a developer who is used to this philosophy is never surprised by having these methods. But developers new to it might be nevertheless.

4.7 Internal Module Design Principles

4.7.1 Information Hiding/Encapsulation (IH/E)

Variants and Alternative Names

- Parnas' Law[17]

Principle Statement

Modules should be encapsulated.

Description

Information hiding and encapsulation are sometimes seen as one and sometimes as two separate but related notions [53][62][63]. This varies through literature. There are three stages of information hiding/encapsulation which can be defined as having a capsule, making the capsule opaque, and making the capsule impenetrable.

Having a capsule means that an object has methods which enable the client of the module to use it without accessing its internal data structures. Making the capsule opaque means that the inner workings are hidden from the clients. This is typically done by using access

modifiers (private, protected). Lastly making the capsule impenetrable means that no client should be able to get a direct reference to an internal data structure.

A properly encapsulated module with an impenetrable capsule is better than an module with just an opaque capsule. And this is better than a module with a non-opaque capsule. But at least having a capsule is better than not having one at all.

Rationale

When the inner workings of a module are hidden from the outside, then they can be changed without any other module noticing it. If the interface of the module stays the same, the rest of the system is not affected by the change. So adhering to IH/E prevents ripple effects.

Strategies

- Use the lowest possible visibility for a variable or method
  - Make all attributes private and use getter and setter methods to access them
  - Better also avoid getters and setters
  - Find suitable abstractions for data types and use appropriate methods instead of just getters and setters
- Avoid aliasing problems with value objects
  - If the programming language supports that use call-by-value objects (like stack objects in C++, structs in C#, records in Delphi, etc.) for value objects like `Date`, `Money`, `EMailAddress`, `TelephoneNumber`, etc.
  - Otherwise use immutable objects which are handled call-by-reference but needn't be copied
- Avoid aliasing problems with lists and similar data structures
  - Copy internal list objects before returning them or only return a read-only `interface` to them

Caveats

See section contrary principles.

Origin

David Parnas: *On the Criteria To Be Used in Decomposing Systems into Modules* [53]

Evidence

Accepted: Virtually every book on object-orientation (e. g. [12]) explains IH/E to some extend.

Relations to Other Principles

Generalizations

Specializations

Contrary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): Not adhering to IH/E is often easier.

Complementary Principles

- →4.4.1 *Model Principle* (MP): IH/E demands having an interface for a module which hides the inner workings. MP tells how such an interface can look like.
- →4.7.3 *Liskov Substitution Principle* (LSP): For subclasses you can waken encapsulation by having a wider `protected` interface which can be used by subclasses. For these cases LSP has to be considered, too.
- →4.5.1 *Tell, don't Ask/Information Expert* (TdA/IE): Encapsulation is about not having getter methods returning constituent internal parts of a module. TdA can be another reason for that.
- →4.5.2 *Low Coupling* (LC): Higher forms of couplings (especially content couplings) break encapsulation.
- →4.7.4 *Principle of Separate Understandability* (PSU): IH/E is about constructing a module in a way that hides the inner workings so it can be used without knowing them. PSU on the other hand is about constructing a module such that its inner workings (and its usage also) can be understood without knowledge about *other* modules.
- →4.6.1 *Easy to Use and Hard to Misuse* (EUHM): A module should be properly encapsulated in order to make it easy to use and hard to misuse.

Examples

Example 1: Date and Tine In Delphi there is the data structure `TDateTime` which represents a specific date and time value [64]. This is an alias name for a double value where the integer part represents the number of days since December 30, 1899. And the fractional part represents the time of day. This alone is a data structure but it is not encapsulated.

The Delphi runtime library (RTL) now specifies functions which operate on `TDateTime` structures. This is "having a capsule". But since it is still possible to access the internal representation directly, the inner workings are not hidden.

This is different in Java. Here the inner workings are hidden. It is not possible to access the private attributes of `java.util.Date`[24]. Here the capsule is opaque (and impenetrable).

Example 2: Aliasing A typical example for an opaque but penetrable capsule is the following:

```
class SomeClass
{
    private SomethingDifferent innerObject;

    public SomethingDifferent getInnerObject()
    {
        return innerObject;
    }
}
```

In such a case the `innerObject` is private, which means it is hidden. But it is revealed by the getter method. In order to establish an impenetrable capsule, the object has to be copied:

```
class SomeClass
{
    private SomethingDifferent innerObject;

    public SomethingDifferent getInnerObject()
    {
        return innerObject.clone();
    }
}
```

4.7.2 Invariant Avoidance Principle (IAP)

Variants and Alternative Names

Principle Statement

Avoid Invariants and Preconditions.

Description

Methods typically have preconditions. Something that has to be true prior to invoking the method so it can work properly. Typical cases are parameters that may not be `null` or have to be in a certain range. A solution is better the fewer preconditions there are.

Furthermore there are (class) invariants, i. e. conditions that have to be true in all observable states during the whole lifetime of an object. Typical invariants are attributes that may not be `null` or have to be in a certain range, lists that have to contain certain objects with certain properties, etc. A solution s better the fewer invariants there are.

While preconditions and invariants are absolutely necessary, introducing further ones comes at a certain cost.

Note that this principle does not apply to loop invariants, control-flow invariants, etc. as there is normally no chance to avoid them. But there can be fewer or more class invariants depending on the solution.

Rationale

A typical kind of defect is the violation of an invariant or a precondition. The more preconditions and invariants there are, the more possibilities there are to introduce defects. And according to →4.3.1 *Murphy's Law* (ML) these possibilities will sooner or later result in defects. So it is better to avoid preconditions and invariants as this reduces the number of potential faults in the software.

Strategies

- If the language supports that, use references which cannot be `null`
  - In C++ use references instead of pointers (see example 3: C++ references)
  - In Java use primitive types instead of their object wrappers (`int` instead of `Integer` but not `int` instead of `Customer`)
- Use value objects instead of primitive types (see example 2: string preconditions)
- Avoid duplication of information. If the same information is stored in different places (maybe in different formats), the values may get out of sync (see also →4.3.4 *Don't repeat Yourself* (DRY)). This also applies to caching.

Caveats

Keep in mind that preconditions and invariants are absolutely necessary for every software. So this principle is constantly violated. Introducing preconditions and invariants is often also done deliberately in order to simplify the code (see →4.3.2 *Keep It Simple Stupid* (KISS)). So the purpose of this principle is mainly to point out that there are drawbacks. By no means invariants are problematic themselves or should be entirely avoided. They just also have disadvantages.

See also section contrary principles.

Origin

This principle is newly introduced here.

Evidence

- Proposed

Relations to Other Principles

Generalizations

- →4.3.1 *Murphy's Law* (ML): ML states that an invariant will eventually be broken. So IAP is the application of ML to invariants.

Specializations

Contrary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): Adding an invariant typically makes the code easier, as it can be assumed that the invariant holds. In fact that is often the very purpose if introducing invariants: Either they make the design easier or they are inevitable. Otherwise they should be avoided.

Complementary Principles

- →4.7.1 *Information Hiding/Encapsulation* (IH/E): When an invariant cannot be avoided, it should at least be encapsulated.
- →4.7.3 *Liskov Substitution Principle* (LSP): Not only the pure number and strength of invariants is relevant. The question is also which types in an inheritance hierarchy should have which invariants. Deriving `Square` from `Rectangle` for example adds an invariant in the subclass. LSP adds another point of view to this problem.
- →4.3.4 *Don't Repeat Yourself* (DRY): Duplication of information, like having the same data in different representations or like caching values, creates invariants. So an invariant sometimes is a hidden DRY violation.
- →4.5.2 *Low Coupling* (LC): One type of precondition is that a specific method has to be called prior to another one. This also results in a temporal coupling.

Examples

Example 1: Index Preconditions The first example is about preconditions of a method getting an index as a parameter.

```
public void prettyPrintItem(List<Item> items, int index)
{
  ...
}
```

This method has the following preconditions:

- `items` may not be `null`
- `index` must be greater or equal `0`
- `index` must be lesser than `items.size()`

Compare the following solution:

```
public void prettyPrintItem(Item item)
{
  ...
}
```

This is better as it just has one precondition: `item` may not be `null`

Example 2: String Preconditions Another typical example are string parameters:

```
public void downloadFile(String url)
{
  ...
}
```

This method has the following preconditions:

- `url` may not be `null`
- `url` must contain a valid URL (which is even a quite complicated precondition)

Compare the following method:

```
public void downloadFile(URL url)
{
  ...
}
```

This is better since there is only one precondition: `url` may not be `null`

Example 3: C++ References Compare the following two methods:

```
void prettyPrint(SomeClass * obj)
{
  ...
}
```

```
1 void prettyPrint(SomeClass& obj)
{
  ...
}
```

In the second version `obj` cannot be `NULL` as it is a reference and not a pointer. So there is one precondition less.

Example 4: DRY A class for complex numbers should either store the real and the imaginary part or absolute value and argument but not both. If both are stored, there is the invariant that both representations result in the same complex number.

So it is better to store just one representation (e. g. the real and imaginary values) and if the other representation is needed (in this case the polar form), it can be computed. This can also be done transparently in the getter method.

Example 5: Caching All forms of caching and redundancy are typical violations of IAP. They are done in order to increase performance. But there is always the disadvantage that all copies have to be kept in sync as there is the invariant that the data may not be inconsistent throughout the copies. There are forms of caching where temporary inconsistencies are tolerated. This is slightly better in terms of IAP but nevertheless there are these consistency constraints and there is the danger of violating them, so to some degree the disadvantage is always there.

4.7.3 Liskov Substitution Principle (LSP)

Variants and Alternative Names

Principle Statement

"Subtypes must be substitutable for their base types." [9, p. 111]

Description

Object-oriented programming languages allow to derive subtypes from base types and subtype polymorphism allows to pass an object of a subtype where ever an object of the supertype is specified. Suppose `P` and `Q` are types (i. e. classes or `interfaces`) and `Q` is derived from `P` (so `Q` is the subtype and `P` is the base type or supertype). A method `m` requiring a parameter of type `P` can be called with objects of type `Q` because every object of type `Q` is also an object of type `P`. This is always true as typically object-oriented programming languages are constructed in that way.

But the programming language does not enforce that the subtype also behaves like the supertype. Method `m` may work with an object of type `P` but not with an object of type `Q`. LSP demands that a subtype (`Q` in the example) has to be constructed in a way that it behaves like the supertype if it is called through the supertype interface. `Q` may have further methods and it may do additional things not observable by `m` but `m` shall be able to safely assume that its parameter behaves like an object of type `P` with respect to all observable state.

Rationale

Let `P` and `Q` be types and `Q` a subtype of `P`. If LSP is not adhered to, there is an operation accessible through the interface of `P` which behaves differently when called on `Q`. So code which is written in terms of `P` will not expect the behavior and will not work as desired.

Strategies

- Only strengthen invariants in subclasses; never weaken them
- Only waken preconditions when overriding methods
- Only strengthen postconditions when overriding methods
- Use Delegation instead of Inheritance
- Figure out better abstractions

Caveats

See section contrary principles.

Origin

Barbara Liskov: *Data Abstraction and Hierarchy* [65]

Evidence

- Examined LSP describes an effect created by object-oriented type systems. There is no human factor in there, so experiments are not needed. The effect was described and thoroughly examined by Barbara Liskov and Jeanette Wing[66]. Their reasoning is presented in section rationale in a simplified form.
- Accepted LSP is widely known in practice, mainly because it is part of Robert C. Martin's SOLID principle collection [9].

Relations to Other Principles

Generalizations

Specializations

Contrary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): Not adhering to the LSP can be easier.

Complementary Principles

- →4.4.1 *Model Principle* (MP): MP demands inheritance relations to resemble an "is-a" relationship. This means that an object of the subclass is also an object of the superclass. This is always true in a technical sense as this is how object-oriented programming languages handle inheritance hierarchies. However MP demands that is shall be true in the model, too. This is slightly different from LSP which rather is about a "is-substitutable-for" relationship.
- →4.7.4 *Principle of Separate Understandability* (PSU): When building inheritance hierarchies, LSP constrains how subclasses are constructed. Namely they should comply with the superclass contract. PSU on the other hand demands that the superclass shall be separately understandable, which means that knowledge of concrete subclasses and their needs should not be necessary to understand the superclass. So a superclass should not have a specific functionality, etc. just because a particular subclass needs this. In contrast to that the superclass of course may provide `protected` features for subclasses in general. But it should be inherently clear that subclasses in general may need this functionality without looking at a particular one.

Examples

4.7.4 Principle Of Separate Understandability (PSU)

Variants and Alternative Names

Principle Statement

Each module shall be understandable on its own – without knowing anything about other modules.

Description

PSU means that:

- By looking at the public methods of a class it should be clear why they are there. That means there should be no method that is only there because a specific other module needs it.
- By looking at the implementation of a module it should be clear how it works and why it was done that way. That means there should be no code that is solely there in order to make another module work.
- By looking at a private method it should be clear what it does. That means there should be no (private) method that is only meaningful in the context of another method.

Rationale

When a module is separately understandable, it is easier to maintain, as no other modules have to be considered during maintenance. It is furthermore more testable, as a unit test can easily test only this particular module without requiring integration with other modules.

Another point of view is that a violation of PSU either means that a part of the functionality does not belong to that module or the module has the wrong abstraction. So this is a sign of a design that needs improvement.

Strategies

When a module does not comply with PSU, this means that either a part of the functionality of the module does not belong here or the module has the wrong abstraction. So strategies for making a solution more compliant with PSU are:

- Move the conflicting functionality to another module where it fits better (see →4.5.1 *Tell don't Ask/Information Expert* (TdA/IE), →4.4.2 *High Cohesion* (HC), and →4.4.1 *Model Principle* (MP)).
- Build up a new module for the conflicting functionality (see →4.4.2 *High Cohesion* (HC)).

- Find the right abstraction for the module that allows the functionality to stay here (see →4.4.1 *Model Principle* (MP)).

Caveats

See section contrary principles.

Origin

This principle is newly proposed in this wiki. Nevertheless it is believed that it is not "new" in the sense that its a new insight. Its rather something that is commonly known but hasn't been expressed as a principle, yet.

Evidence

- Proposed (see origin)

Relations to Other Principles

Generalizations

Specializations

Contrary Principles

- →4.3.2 *Keep It Simple Stupid* (KISS): Not to adhere to PSU is sometimes easier.

Complementary Principles

- →4.7.1 *Information Hiding/Encapsulation* (IH/E): PSU is about constructing a module such that its inner workings (and its usage also) can be understood without knowledge about other modules. IH/E on the other hand is about constructing a module in a way that hides the inner workings so it can be used without knowing *them*.
- →4.4.1 *Model Principle* (MP): The model contains the only information that should be necessary to understand the module. And if the abstraction of the model is wrong, MP helps getting it right.
- →4.5.1 *Tell, don't Ask/Information Expert* (TdA/IE): At its heart PSU is about responsibility assignment. When a module is not separately understandable, this means that a responsibility is scattered across several modules. TdA/IE gives another aspect of responsibility assignment.
- →4.5.2 *Low Coupling* (LC): Not adhering to PSU means that responsibilities are scattered across several modules. This typically also means increased coupling.

Examples

Example 1: Parsing Data Suppose a program parses data stored in an spreadsheet file. There are three classes:

- `SpreadsheetReader`: This reads the spreadsheet and creates `DomainObject` objects.
- `DomainObject`: This is the data which was contained in the spreadsheet and is now processed by the program in some way.
- `SpreadsheetWriter`: This class takes a `DomainObject` and writes it back to the spreadsheet.

In such a scenario it might be convenient to simplify `SpreadsheetWriter` by adding information about the spreadsheet to `DomainObject`. This might be some cell coordinates for example. `SpreadsheetReader` can store them into the newly created `DomainObject` and `SpreadsheetWriter` uses the data to store the `DomainObject` to the correct position in the spreadsheet. The problematic method is `DomainObject.getCellPositionInSpreadsheet()`.

This is a simple solution (see →4.3.2 *Keep It Simple Stupid* (KISS)) but it violates PSU. `DomainObject` is not understandable on its own. It holds data (namely the cell position in the spreadsheet) that is only meaningful in the context of the other two modules. During maintenance this data could accidentally be altered (resulting in a corrupted output file). Maintenance effort is also increased simply by distracting the maintainers who might wonder what this data is and if it is relevant for their task.

A better solution (w. r. t. PSU) would be to give `SpreadSheetWriter` the ability to determine the correct position in the spreadsheet itself. This is more complicated and may involve searching the spreadsheet for the correct position. But `DomainObject` is easier to understand and less prone to errors.

Example 2: Dependent Private Methods In a module that computes results in a bowling game there might be a method `strike()` which returns true when the player has thrown a strike, i. e. hit all 10 pins with only one ball throw.

```
private int ball;
private int[] itsThrows = new int[21];

private boolean strike()
{
    if (itsThrows[ball] == 10)
    {
        ball++;
        return true;
    }
    return false;
}
```

Here the method not only computes if the current throw is a strike or not but also advances the counting variable `ball`. This is only meaningful in the context of another method. If this is correct behavior or a defect cannot be told solely by looking at this method. Should ball be increased by 1 or 2? Should it also be increased when the throw is not a strike? Should it be increased at all? It cannot be told without looking at other parts of the code. So this method violates PSU.

The following solution is better:

```
private int rolls[] = new int[21];

private boolean isStrike(int frameIndex)
{
    return rolls[frameIndex] == 10;
}
```

Here no counting variable is increased in some way. Furthermore this method does not rely on a correctly set private variable but gets a parameter.

This example is taken from Robert C. Martin.

- First version: see [9]
- Second version: see [67]

5 Discussion of the Principle Language

5.1 Usage

5.1.1 Usage of Principle Languages

The general idea behind principles and how they are used has already been presented in chapter 2. In a nutshell it's this: When making a design decision, possible solutions can be assessed using a principle language. A principle language interconnects principles so that the consideration of one principle inevitably leads to other principles that should also be considered. Starting from one or two obvious principles, typically results in a set of around four characterizing principles that describe the design problem by pointing out advantages and disadvantages of different solutions. The solutions can then be informally rated according to the principles and the designer can make an informed decision. In this way the reasons for taking the design decision are also easily communicable as they manifest in the principles. The set of principles then becomes a language for thinking and talking about design.

When to make design decisions and which principles to start with, heavily depends on the software development process. In particular waterfall-like processes and agile processes work fundamentally different. Especially there are different generative design approaches (see section 2.1.1) in place.

Furthermore the usage depends on the principle language or rather the purpose or level of abstraction of the principle language. This thesis presents a principle language for object-oriented design. But other principle languages may be envisioned, too. There may be principle languages more focused to implementation, more to architecture or to requirements. These differences in the level of abstraction may also influence the usage. The following sections describe how this particular principle language presented in chapter 4 can be used in traditional, plan-driven environments and in agile environments.

5.1.2 Navigating the Principle Language

In order to end up with the aforementioned set of characterizing principles, one has to start with one or two principles and then navigate the principle language. The starting principles are often obvious as they are directly about the given design problem but they can also be suggested by the used generative design method (see sections 5.1.4 and 5.1.5).

When designing inheritance hierarchies the →4.7.3 *Liskov Substitution Principle* (LSP) is the obvious starting principle, for deciding on whether and how to remove duplicated code →4.3.4 *Don't Repeat Yourself* (DRY) is the obvious choice and when deciding upon the signature of a method, it's →4.6.1 *Easy to Use and Hard to Misuse* (EUHM). LSP is

precisely about inheritance hierarchies, DRY about duplication and EUHM about interface design. So these are the obvious choices.

But as the suitable starting principle is not always as evident as in these examples, there is further aid in finding it. The nineteen principles in the principle language are roughly categorized. There are general principles, principle for modularization, module communication, interface design, and internal module design. So when facing a design problem about how to divide a system into modules, how these modules shall communicate, how module interfaces shell be crafted or how the internal structure of a module shall look like, there is a small group of principles being candidates for a starting principle. Moreover the very first of the principles in these groups (ML, MP, TdA/IE, EUHM, IH/E) is slightly more general than the others making it a predominant candidate for a starting principle.

When one or two starting principles have been found, the principle language lists other principles that are likely to be considered, too. There are different relationships between the principles. First of all there are the generalization/specialization relationships. Sometimes a principle does not quite fit but somehow resembles the idea. In this case it may be helpful to replace it with a generalization of the principle. On the other hand there are cases where a specialization fits better because it is more tailored to the given problem. A more specialized principle typically is more helpful as it is easier to apply when it fits the problem. More general principles are harder to apply but are applicable to a wider range of design problems.

Normally generalizations and specializations are meant to replace the principle currently considered. But there are also cases where one might want to keep both in the characterizing set. This is also the usual case for the other two relationships: contrary and complementary principles. These are not meant to point to possible replacements but to principles that should be considered in addition. Contrary principles are more likely that they reveal drawbacks of the design so they can be regarded more important. Complementary principles rather point to further aspects of the design problem that are not necessarily disadvantages. This categorization is only rough. For a particular design problem a contrary principle may be rather complementary and a complementary principle may also reveal drawbacks.

The principle discovery is recursive which means that the newly discovered principles are again examined for related principles that may also become part of the characterizing set. This process stops long before a major part of the principle language is considered because principles that do not qualify for being included into the characterizing set are not examined further.

A principle that is somehow related to a principle that is currently considered is not guaranteed to be relevant for the given design problem. In the majority of the cases it is clear if a principle qualifies for consideration. In those cases where it is not obvious or the designer is too inexperienced to judge that, the principle language gives further guidance. First there is a short textual explanation of the relationship. This explains the most common reasons for navigating to the other principle. Furthermore every principle description has a rationale section. If the rationale given there also applies to the given design problem, the principle qualifies and is inserted into the characterizing set. Also the section description and examples may help understanding the principle.

This discovery process is not meant to last long. It's neither meant to be documented

or otherwise to be followed pedantically. The idea is to have a lightweight approach which is not much more than a way of thinking. While inexperienced designers might want to look every principle up, more advanced ones will only need a short glance at the principle language graph (see figure 4.1 on page 24). Experienced designers who do not need this kind of guidance at all can skip all that and just use the principle language as a set of vocabulary for talking about their design. So the principle language helps designers with very different levels of experience.

5.1.3 Level of Abstraction

Concerning the level of abstraction, the principle language is used for all low-level design tasks. The language is not designed for large-scale architectural decisions. If a principle language shall also be used for that purpose, another principle language will be necessary. For such a case some low-level principles like IAP are not helpful and others concerned with groups of classes, packages, layers, and subsystems are missing. But everything beneath that level, everything involving one or a few classes, can be addressed.

Everything which takes place inside a single method is also out of scope. The principle language does not fit for algorithm design and coding. A completely different set of principles would be necessary here. Principles on this scale would rather deal with the usage of particular language constructs like loops, recursions, and try-catch blocks.

5.1.4 Using Principle Languages in a Plan-Driven Environment

Traditional, plan-driven processes are typically derived from the waterfall-model. This means there is a dedicated design phase and a subsequent implementation phase. The number of phases, the degree of design documentation, the produced artifacts, etc. may vary but the essential way of thinking is that design always precedes implementation.

Depending on how detailed the design phase is, there may still be some design decisions left to the implementation phase. The one extreme case would be that only the coarse-grained architecture is designed in the design phase leaving everything else to implementation. And the other extreme would be that absolutely everything is designed upfront so that implementation is only a manual task transforming the specification into program code.

Independent from the specific phase there are certain low-level design decisions to be taken and they are made before they are implemented. So the principle language is used in the design and implementation phases for all the design tasks at the appropriate level of abstraction (see section 5.1.3).

The typical generative approach used here is the Booch method [48] or something similar. This means the reality is modeled and this object model of reality is gradually transferred to an executable program. The principle directly corresponding to that idea is the →4.4.1 *Model Principle* (MP). So in many cases the starting principle will be MP. Depending on the given design problem sometimes another principle is the more obvious start. But in plan-driven approaches there is a tendency for MP.

5.1.5 Using Principle Languages in an Agile Environment

Agile processes work fundamentally different. Normally there is no dedicated design phase preceding implementation and if there is one, it concentrates on important architectural decisions. The generative design approach which is typically used is test-driven development (TDD) (see [9] or any other book on agile methods). This means, software development is the continuous repetition of the following steps:

1. Write a test case (i. e. an automated test) for the next bit of functionality.
2. Write just enough code to make the test pass.
3. Refactor in order to improve the structure of the software.

In practice there is more about TDD than that but this is the part that is important for the discussion here.

Each step involves a bit of design. The design decisions taken in these steps are quite different so there are different preferred starting principles for each step: The first step is to write a test case for the next piece of functionality. This already involved the design decisions dealing with interface design. The test case is written against a yet to develop interface. Method identifiers and their signatures, usage patterns and everything that determines how a module is used is decided in this phase. The main question the developer asks here is "How would I want to use this module?". The corresponding principle is →4.6.2 *Principle of Least Surprise* (PLS). So PLS is the most likely starting principle during this step.

In the second step, the implementation, TDD demands doing the simplest thing that makes the test pass. This is basically →4.3.2 *Keep It Simple Stupid* (KISS). First the implementation shall be simple and just make the test pass. Later tests will ensure that the functionality gets more and more complete. So for the second step the KISS principle is most likely the starting principle.

The purpose of refactoring is to improve the design. In the previous step there was a strong tendency towards simplicity. But only focusing on simplicity is harmful for good design. Furthermore as there is no upfront design phase combining thoughts about several features, the implementation may diverge further and the design degrades. So refactoring is needed in order to ensure that the design quality stays high. Typical starting principles in this phase are →4.3.4 *Don't Repeat Yourself* (DRY), →4.5.2 *Low Coupling* (LC), and →4.4.2 *High Cohesion* (HC). But almost every other principle can also be the starting point, depending on where which design problem to cure. To further aid refactoring, each principle description also lists a number of strategies that hint how a design which better conforms to the principle may look like. This is not a replacement for refactoring procedures as they are described in [8] because these strategies are more general and sometimes rather describe a desired structure than a procedure for transforming the code into that structure. So the strategies are rather hints than refactorings and [8] complements the principle language.

A second difference between agile and plan-driven application of the principle language is the weighting. The principle language does not tell which principles are more important

than others. Judging this is still the task of the designer and it is also highly dependent on the concrete requirements for the software. Nevertheless agile designers will feel a tendency for →4.3.2 *Keep It Simple Stupid* (KISS) and against →4.3.5 *Generalization Principle* (GP). The reason for that is that speculative design, meaning the preparation of the design for possible future enhancements, is objected to by agile methodologies.

Apart from purely plan-driven waterfall life-cycles and purely agile processes, there are many intermediate approaches ranging from waterfalls with agile elements to agile processes with additional documentation and design phases. The usage of the principle language will depend on the concrete arrangement of the processes.

5.2 Evidence

One important question which needs to be addressed is which evidence do we have that the 19 principles are valid. There are sound reasons to believe in all of them (documented in the rationale section of each principle description). But there may also be further evidence. There can be documented case studies, experiments, empirical studies, etc.

Each principle has been categorized in order to answer that question (see table 5.1). Three principles are completely new (PSU, IAP, UP). For these there is naturally no evidence despite their rationale and the belief that they are helpful. MIMC is also new but there is already some research on certain aspects of the principle.

Most of the principles—thirteen of nineteen—are already accepted in practice. This means that there are at least certain communities of practitioners believing in them and using them. Some of these "accepted" principles are broadly known and have become something like folklore of software development (e. g. Murphy's Law (ML) or the KISS principle) or are already central principles to design (e. g. →4.5.2 *Low Coupling* (LC) and →4.4.2 *High Cohesion* (HC)). Others are only known to certain groups of people. The →4.3.6 *Rule of Explicitness* (RoE), which states that "explicit is better than implicit" is well known among Python developers but hardly known to others. Nevertheless these principles are *accepted*. Although they might not have been subject to scientific research yet, practice shows that they are perceived to be helpful.

Scientific research has currently only examined seven of the principles. And even for those principles where there is certain research, this often concentrates on parts and aspects of the principles leaving several open questions. So research lags behind here. Further research is necessary to affirm the validity of the principles.

The last column of the table refers to the fact that there may be some doubt about the principles. There are always drawbacks and each principle focuses only on one aspect neglecting the others. This is the reason why there are contrary principles. But there may also be doubt about the positive effect the principles claim to have. This doubt threatens the validity of the principles. Only two principles of the principle collection are marked questioned: MIMC and MP.

For MIMC ("more is more complex") this is the case because there is debate about the so-called "Goldilocks Conjecture" which can be seen as a corollary of MIMC. If principles

Table 5.1: Evidence categorization for the principles of the principle language

Principle	*Accepted*	*Examined*	*Questioned*
ML	✓		
KISS	✓	✓	
MIMC		✓	✓
DRY	✓	✓	
GP			
RoE	✓		
MP	✓		✓
HC	✓	✓	
ECV	✓	✓	
TdA/IE	✓		
LC	✓	✓	
DIP	✓		
EUHM			
PLS	✓		
UP			
IH/E	✓		
IAP			
LSP	✓	✓	
PSU			

were laws, mathematical theorems or otherwise hard rules, an invalid corollary would mean that MIMC would have to be considered invalid, too. But as principles are only meant to be helpful rules of thumb or heuristics, this is not a problem.

The reasons why the Model Principle (MP) is questioned are more complicated. Basically object technology as a whole or at least its domain-modeling aspect which is central is criticized (for more details see the corresponding principle description in section 4.4.1). As it is the goal of this thesis to construct a principle language for object-oriented design it is not the question whether or not to use object technology. So these doubts are mainly out of scope here.

Summarizing this there is enough evidence for the principles so they can be incorporated into the principle language. Most of them are already accepted in practice and only few are really new. Nevertheless science has not caught up here and further work is still necessary.

5.3 Other Principle Collections

As already showed in section 2.3 there are already several other principle collections. The aforementioned section compares the design approaches (or explains the lack of those). Here the principle collections are compared to the principle language by comparing the included principles. The mentioned collections are: The SOLID principles by Robert C. Martin [9], the GRASP principles by Craig Larman [12], the tips in "The Pragmatic Programmer" by Andrew Hunt and David Thomas (abbreviated "PragProg" in this section) [13], the principles collected by Bertrand Meyer in his book "Object-Oriented Software Construction" (OOSC) [14], the Unix philosophy principles as stated by Eric S. Raymond in "The Art of Unix Programming" [15], "201 Principles of Software Development" by Alan M. Davis [16] and the laws, hypotheses, and conjectures discussed in "A Handbook of Software and Systems Engineering" (abbreviated just "Handbook" here) by Albert Endres and Dieter Rombach [17].

Two questions can be asked with regard to such a comparison: a) Are principles missing in the principle language that are included in other principle collections? This would mean the principle language needs to be enhanced in order to make it applicable to a broader range of design decisions. b) Are there any other principle collections that already include the principles of the principle language. This would mean that the principle language is superfluous as it would not add anything new despite the approach discussed in chapter 2.

Table 5.2 shows how principles from other principle collections are represented in the principle language presented in chapter 4. ✓ means the principle is directly part of the principle language, (✓) means the principle is included in a generalized or specialized form, ✗ means the principle is not included, and — means it cannot be included because it is not a principle in the sense discussed here.

Table 5.2: Comparison to Other Principle Collections 1

Principle	*Represented?*	*Comment*
SOLID		
SRP	(✓)	HC is a generalization
OCP	(✓)	ECV is a generalization
LSP	✓	
ISP	(✓)	HC is a generalization
DIP	✓	
GRASP		
Controller	✗	rather architectural scale
Creator	✗	MP compensates for the lack of this principle
High Cohesion	✓	
Indirection	—	not a principle
Information Expert	✓	

continued on the next page

Low Coupling	✓	
Polymorphism	—	not a principle
Protected Variation	—	not a principle
Pure Fabrication	—	not a principle
The Pragmatic Programmer		
Don't Repeat Yourself (tip 11)	✓	
Make It Easy To Reuse (tip 12)	(✓)	GP is a generalization
Eliminate Effects Between Unrelated Things (tip 13)	✗	very general principle; compensated by HC, ECV, PSU, and IH/E
Program Close To The Problem Domain (tip 17)	✗	very special principle concerned with the use of deomain specific languages; compensated by GP and MP
Keep Knowledge in Plain Text (tip 20)	✗	very special principle; compensated by KISS and GP
Write Code That Writes Code (tip 29)	(✓)	DRY is a generalization
Crash Early (tip 32)	✗	rather on implementation scale
Use Assertions to Prevent the Impossible (tip 33)	✗	rather on implementation scale
Use Exceptions for Exceptional Problems (tip 34)	✗	rather on implementation scale
Finish What You Start (tip 35)	✗	rather on implementation scale
Minimize Coupling Between Modules (tip 36)	✓	
Configure, Don't Integrate (tip 37)	✗	rather on architectural scale
Put Abstractions In Code, Details In Metadata (tip 38)	✗	rather on architectural scale
Always Design for Concurrency (tip 41)	✗	very special principle; compensated by GP
Separate Views From Models (tip 42)	✗	rather on architectural scale
Abstractions Live Longer than Details (tip 53)	✓	this is basically GP
(54 other tips)	—	not a principle
Object-Oriented Software Construction		
Direct Mapping	✓	MP
Few Interfaces	(✓)	LC is a generalization
Small Interfaces	(✓)	LC is a generalization
Explicit Interfaces	(✓)	RoE is a generalization
Information Hiding	✓	

continued on the next page

Linguistic Modular Units	✗	rather implementation scale
Self-Documentation Principle	✗	compensated by PSU
Uniform Access Principle	✗	very special principle; compensated by IH/E
Open-Closed Principle	(✓)	ECV is a generalization
Single Choice Principle	(✓)	DRY is a generalization
Command-Query Separation	(✓)	PLS is a generalization
Operand Principle	✗	very prescriptive principle; compensated by MIMC, HC, and MP
Symbolic Constant Principle	✗	rather implementation scale
Taxomania Rule	✗	compensated my MIMC
(186 other principles, rules, precepts and definitions)	—	not a principle
Unix Philosophy (Eric S. Raymond)		
Rule of Modularity	✗	rather architectural scale
Rule of Clarity	(✓)	KISS is a generalization
Rule of Composition	✗	rather architectural scale
Rule of Separation	✗	rather architectural scale
Rule of Simplicity	✓	KISS
Rule of Parsimony	(✓)	KISS is a generalization
Rule of Transparency	(✓)	KISS is a generalization
Rule of Robustness	(✓)	KISS is a generalization
Rule of Representation	✗	not applicable
Rule of Least Surprise	✓	
Rule of Silence	✗	rather UI design
Rule of Repair	✗	rather implementation scale
Rule of Economy	✗	not applicable
Rule of Generation	(✓)	DRY is a generalization
Rule of Optimization	✗	rather process design
Rule of Diversity	✗	very general principle
Rule of Extensibility	(✓)	GP is a generalization
201 Principles of Software Development		
Encapsulate	✓	IH/E
Don't Reinvent the Wheel	✗	very specific principle; compensated by others depending on context
Keep It Simple	✓	KISS
Avoid Numerous Special Cases	✗	rather implementation scale
Minimize Intellectual Distance	(✓)	MP is a specialization

continued on the next page

Keep Design Under Intellectual Control	✗	compensated by KISS, IH/E, PSU, MP, and PLS
Maintain Intellectual Integrity	(✓)	UP is a generalization
Conceptual Errors Are More ignificant Than Syntactic Errors	✗	rather process design
Use Coupling and Cohesion	✓	HC, LC
Design for Change	(✓)	GP is a generalization
Design for Maintenance	(✓)	
Design for Errors	✓	ML
Build Generality into Software	✓	GP
Build Flexibility into Software	(✓)	incorporated in GP
"Garbage In, Garbage Out" Is Incorrect	✗	rather implementation scale
(11 other principles)	—	not a principle
A Handbook of Software and Systems Engineering		
Simon's Law	✗	rather architectural scale; partly addresses by LC
Constantine's Law	✓	Constantine's Law is the combination of LC and HC
Parnas' Law	✓	IH/E
Fitts-Shneiderman Law	✗	only relevant for UI design
Booch's Second Hypothesis	(✓)	incorporated in MP
Bauer-Zemanek Hypothesis	✗	rather process design
Gamma's Hypothesis	✗	very special principle; lack compensated by several others depending on context
Dijkstra-Mills-Wirth Law	✗	very general principle; compensated by MP, KISS, PSU, and PLS
McIlroy's Law	✗	very special principle; lack compensated by several others depending on context
Dahl-Goldberg Hypothesis	✗	not applicable
Beck-Fowler Hypothesis	✗	rather process design
Lehman's First Law	(✓)	GP is a generalization
Lehman's Second Law	✗	rather process design
McCabe's Hypothesis	✗	rather process design
Wilde's Hypothesis	✗	not applicable
(11 other laws, hypotheses and conjectures)	—	not a principle

The table shows that most of the principles in the other collections are either incorporated or do not fit into the principle language. In th latter case they belong to other contexts (high-level architecture, UI design, process design, etc.) or they are not even principles according to definition 5. Only a few principles are not included because they are very

Table 5.3: Comparison to Other Principle Collections 2

Principle	*SOLID*	*GRASP*	*PragProg*	*OOSC*	*Unix*	*201 Principles*	*Handbook*
ML						✓	
KISS					✓	✓	
MIMC							
DRY			✓	(✓)	(✓)		
GP			✓		(✓)	✓	(✓)
RoE				(✓)			
MP				✓		(✓)	(✓)
HC	(✓)	✓				✓	✓
ECV	(✓)			(✓)			
TdA/IE		✓					
LC		✓	✓	✓		✓	✓
DIP	✓						
EUHM							
PLS					✓		
UP							
IH/E				✓		✓	✓
IAP							
LSP	✓						
PSU							

special or very general. In these cases other principles compensate for the lack so there is no need to enhance the principle language. Question a) from above can thus be answered with no. At least these principle collections do not indicate that principles are missing.

Table 5.3 depicts the inverse relationship: Which principles of the principle language in chapter 4 are represented in which other principle collection? ✓ means the principle is represented and (✓) means a generalization or specialization is included.

The table shows that none of the collections includes not even half of the principles. Also question b) from above can be answered with no. These principle collections do not make the principle language obsolete. It contains a unique set of principles covering a large part of the design space.

6 Evaluation

6.1 Goals

When something new is proposed, it needs to be evaluated. Such evaluations can be very laborious and can be subject of dedicated theses. The concept of principle languages is very new as this is a new area without prior research. Nevertheless at least two small experiments have been conducted in order to evaluate the approach and the principle language.

The first experiment is the prototypical development of a feed reader application using an agile process. As explained in section 5.1.5 in agile processes the principle language is used to initiate refactoring or rather to judge whether refactoring is necessary. The two competing approaches for that purpose are Robert C. Martin's SOLID [9] and Martin Fowler's Code Smells [8] (see sections 2.3.2 and 2.3.3). So for the experiment the usage of the principle language has been documented and compared to the two competing approaches. The second experiment is the examination of the CoCoME system [68] using the principle language.

The goals of this experiment are to evaluate the approach as well as the principle language. First of all the question is whether principle languages can be used as described in chapter 2. And second it needs to be evaluated if the principle language presented in chapter 4 is helpful or needs rework.

6.2 FeedReader

6.2.1 Setup

The task in this experiment was to develop a feed reader for RSS news feeds [69]. Such a software downloads and parses XML files and presents the output. The software uses Java, Swing and XML. As a development methodology, a agile approach using test-driven development (TDD) was used.

The FeedReader example is only a very small, prototypical piece of software. Nevertheless it yielded 49 design decisions. Each of these decisions was taken using the approach proposed in this thesis. The characterizing sets of principles and the decisions were documented. Furthermore it was also documented how much guidance Fowler's code smells and Martin's SOLID principles would give.

6.2.2 Questions

For the FeedReader experiment the following questions have been addressed:

1. Does the approach work, i. e. does the principle language generate appropriate characterizing sets of principles for arbitrary design problems?
2. Are the characterizing sets small enough, i. e. typically not more than fife principles? Too large characterizing sets become unhandy.
3. Are principles missing, i. e. are important aspects not considered? In such a case the principle language would need to be enhanced.
4. Are some principles unused (and should thus be removed)?
5. Does the principle language help better than SOLID and code smells?

Naturally it is hypothesized that the approach does work and the principle language is neither too small nor too large. SOLID has the fewest principles hence it is assumed that the other two approaches will perform better in the experiment. Presumably the principle language approach is applicable for a widest set of design decisions and gives the most guidance. The experiment will show if these assumptions are correct.

6.2.3 Results

The experiment showed that the principle language is able to produce characterizing sets for arbitrary design decisions (see question 1). The protocol in appendix A lists for each design decision how principle discovery was done. Some examples:

Design decision #3 is a very typical low-level design decision made in the refactoring step: There is some duplicated code. The developer now has to decide whether to extract this duplicated code into a new method or not. The obvious starting principle is →4.3.4 *Don't Repeat Yourself* (DRY) as this principle is exactly about avoiding duplication. In this case this is quite similar to the code smell "duplicated code". But the principle language does not limit itself to one aspect. Avoiding duplication comes at a cost and the principle language lists these disadvantages as contrary principles so the developer can make an informed decision. In this case DRY lists →4.3.2 *Keep It Simple Stupid* (KISS) as a contrary principle. Not introducing a further principle may be simpler. So at this stage the characterizing set is {DRY, KISS}. KISS in turn lists →4.3.3 *More Is More Complex* (MIMC) as a specialization. Here MIMC fits better as it more specifically describes the disadvantage: A new method would have to be added which increases complexity as it increases the number of methods. MIMC replaces KISS in the characterizing set. As the relationships of MIMC do not qualify for being included into the characterizing set, the result of the principle discovery is {DRY, MIMC}. There are two possible decisions the developer can make: either leave the code duplication as it is (according to MIMC) or extract a new method (according to DRY). This decision is left to the developer but the principle language helps finding the relevant aspects to consider. Here it was decided to extract a new method because the additional complexity according to MIMC was justified negligible.

Another interesting example would be design decision #9: A prior refactoring step produced a method `parseFeedItem` which takes the parameters `items` and `i`. The method

parses item `i` in the list `items`. Certainly every good developer knows that it is better to supply the item to parse directly instead of specifying a list and an index. An inexperienced developer, like an apprentice for example, might say that it does not matter as both versions do exactly the same thing. But an experienced developer "feels" that the one solution is clearly better than the other. Explaining the reasons for that feeling may be difficult, tough. The principle language helps here.

As starting principles for this decision →4.6.1 *Easy to Use and Hard to Misuse* (EUHM) and the →4.3.5 *generalization Principle* (GP) are used. The design problem is about interface design. So EUHM is a natural choice. GP was added because it was realized that the single parameter solution would allow for any item to be parsed and not just ones contained in a list. EUHM leads to →4.7.2 *Invariant Avoidance Principle* (IAP) and →4.4.1 *Model Principle* (MP). GP leads to the →4.3.6 *Rule of Explicitness* (RoE) which in turn adds MIMC. So the produced characterizing set is {EUHM, GP, IAP, MP, RoE, MIMC}. each of these principles give reasons for preferring the single parameter solution over providing a list and an index:

EUHM The solution with one parameter is easier to use and even more the other solution bears possibilities for misuse as one could specify the wrong index.

GP The single parameter solution is more general as it works with items not stored in a list.

IAP The two parameter solution is worse because it has the precondition that the index has to be inside the bounds of the list.

MP The single parameter solution is better because the "model of the method", i. e. something that parses items, logically does not include a list.

RoE Passing the item directly to the method is more explicit.

MIMC more parameters are worse.

Not every of these reasons is easy to see (e. g. the MIMC reason is clear and the MP reason is less clear). But the principle language revealed plenty of possibilities to explain the apprentice why the single parameter solution is clearly the better choice.

Note that a different path trough the principle language might have been taken, too. Another possibilities would be for example: EUHM leads to IAP and KISS, KISS to GP, GP to RoE, and RoE to MP. KISS is then replaced by MIMC. This path has even the advantage that it only needs one starting principle. And also the characterizing sets are not unique. An equally good characterizing set would be {ML, KISS, GP, RoE}. This set is produced by the the following sequence: EUHM is replaced by ML, ML leads to KISS, KISS to GP, GP to RoE. And there are also several other possibilities. So the approach is neither determined in the produced characterizing set, nor deterministic in how it is produced. Nevertheless it generates reliably generates appropriate characterizing sets and is thus helpful.

Table 6.1 shows the characterizing sets of principles for all the design decisions taken in the development of the FeedReader example. The rightmost column shows the size of the characterizing sets. the size ranges from 1 to 7 principles with an average of approximately

3.2. So question 2 can be answered: The characterizing sets typically are small enough. Large characterizing sets are rare exceptions (there is only one design decision which yielded a characterizing set of 7 principles). So the characterizing set stays small enough to be useful.

Table 6.1: Characterizing Sets of Principles for the Design Decisions Taken in FeedReader

#	ML	KISS	MIMC	DRY	GP	RoE	MP	HC	ECV	TdA/IE	LC	DIP	EUHM	PLS	UP	IH/E	IAP	LSP	PSU	Total
1		✓				✓	✓						✓							4
2		✓					✓													2
3			✓	✓																2
4			✓	✓																2
5	✓	✓	✓	✓			✓													5
6		✓	✓		✓															3
7		✓	✓		✓															3
8			✓																	1
9			✓		✓	✓	✓						✓				✓			6
10															✓					1
11		✓											✓							2
12							✓			✓						✓				3
13		✓				✓	✓						✓							4
14			✓				✓	✓	✓		✓									5
15							✓						✓	✓						3
16											✓		✓	✓						3
17			✓				✓	✓	✓		✓			✓						6
18		✓			✓						✓									3
19													✓	✓						2
20							✓							✓						2
21		✓					✓							✓						3
22		✓		✓																2
23		✓		✓																2
24		✓	✓								✓									3

continued on the next page

	3	2	6	4	2	3	5	5	2	2	4	2	4	2	4	5	2	3	4	2	4	3	2	7	2	
25																			✓							1
26																										0
27			✓			✓												✓								3
28			✓			✓															✓			✓		5
29																										2
30																		✓				✓	✓			10
31			✓				✓		✓						✓			✓								12
32												✓														1
33			✓				✓				✓		✓			✓								✓		11
34			✓			✓															✓					4
35			✓								✓					✓	✓		✓					✓		7
36				✓							✓					✓	✓		✓	✓				✓		9
37			✓	✓											✓	✓	✓				✓			✓		17
38							✓		✓				✓									✓	✓			9
39							✓		✓				✓							✓		✓				11
40					✓																					8
41																										19
42																										23
43																										4
44																										Total
45																										
46																										
47																										
48																										
49																										
Total	4	23	19	8	11	9	17	9	7	4	11	1	12	10	2	5	3	0	1							

Listing 6.1: Subscription Depends on Downloader and Channel

```
public class Subscription
{
    // these need to either reference a concrete class or a dummy:
    // DownloaderImpl or DummyDownloader
    // and ChannelImpl or DummyChannel respectively
    private Downloader downloader;
    private Channel channel;
    ...
}
```

During the course of the experiment one principle was found to be missing and was added (→4.3.6 *Rule of Explicitness* (RoE)). Design decision #31 deals with the mechanism to be used for providing test isolation. There is a class called `Subscription`. For this class a JUnit test shall be written. A desirable property of unit tests is test isolation which means that the class under test is tested without integrating it with other classes. Test isolation ensures that the tests of a module only find defects that manifest in that particular module. Without test isolation a fault in one module may cause a large part of the test cases to fail making it hard to find the actual cause and the piece of code that need to be fixed.

In order to allow for test isolation, the other classes `Subscription` depends on (`Downloader` and `Channel`) need to be replaced by dummies or "stubs". But this cannot be done if `Subscription` directly references the other classes in its code. So it may only reference an `interface` type which the other classes have to implement[1]. So `Downloader` and `Channel` are `interface`s and there are concrete classes `DownloaderImpl` and `ChannelImpl` which implement them. But this is not enough because somewhere in the code the concrete classes have to be instantiated. Furthermore somehow the internal references of `Subscription` have to be updated to in the production code they point to the created classes and in the test code they reference the dummies (see listing 6.1).

There are several possible mechanisms for doing so and a decision is needed which of these mechanisms to implement. In particular there are three patterns or combinations of patterns which would solve the problem. A *Service Locator* [70] can be used, *Dependency Inversion* [70], or a combination of *Abstract Factory* and a loadable variant of the *Singleton* pattern [3].

Essentially dependency injection works by providing the possibility to set the inner references from the outside (via setter methods, constructor parameters, etc.). A service locator or "registry" is an object that can be queried for a service (i. e. an object) of a certain type. in the above situation ti would mean that the service locator would be filled with concrete objects or dummies and `Subscription` would query the locator and get what is stored in there. Implementing the last possibility, the abstract factory, would mean to add an abstract class providing methods for getting the required objects. Two concrete classes are

1 this is also a result of the →4.5.3 *Dependency Inversion Principle* (DIP)

derived, one returning the normal "Impl" classes and the other returning dummies. Abstract factories are often singleton. In order to make the singleton replaceable, there are variants of the patter which allow the singleton to load arbitrary objects of subclasses (original idea from [70] described in [71]).

The essential difference between the service locator and the abstract factory approach is, that the abstract factory loads whole families of classes whereas the service locator stores each object independently. See listing 6.2 for rough sketches on how the solutions would look like.

Each solution has its advantages and disadvantages. One aspect is described by →4.5.2 *Low Coupling* (LC). LC is the obvious starting principle as the goal is to decouple `Subscription` from the rest of the system. The principle language yields →4.3.2 *Keep It Simple Stupid* (KISS) as the next principle and KISS then leads to →4.3.1 *Murphy's Law* (ML) and the →4.3.5 *Generalization Principle* (GP). So the characterizing set is {LC, KISS, ML, GP}.

Clearly the abstract factory approach is the most complicated one, i. e. KISS is against this solution. Furthermore the other two solutions are more general because the abstract factory can only load whole families of classes whereas the others can vary in each service/object independently. So also GP is against this solution. And because this solution is at least not better with respect to MC and ML, it is ruled out leaving only dependency injection and the service locator.

For comparing the two remaining possibilities a further aspect needs to be considered which is explicitness. One disadvantage of a service locator is, that such an object obfuscates the real dependencies. A class which depends on the service locator may depend on every service it provides. Just looking at the interface of a class only reveals the dependency to the service locator (and if it is a singleton not even that) but the real dependencies are hidden. This is different with dependency injection. Here all dependencies are obvious from the interface of the class. On the other hand dependency injection is less explicit with respect to where the dependencies come from. It harder to see where the dependencies are provided as this happens somewhere else. This is especially true when a dependency injection framework like spring [72] is used. In such a case the dependencies are "magically" filled in by the framework making the "wiring" of the objects implicit.

As a result both solutions are more or less equal with respect to KISS, ML and GP. Dependency Inversion provides a slightly lower coupling as there is no dependency to an additional module. It is also better with respect to explicitness of the interface but the service locator is slightly better with respect to explicitness of the "wiring". For the FeedReader example dependency injection (without a framework) was chosen.

The missing principle, →4.3.6 *Rule of Explicitness* (RoE) was added to the principle language. Adding the principle to the language furthermore showed that it is also helpful for other design decisions. The protocol was adjusted accordingly.

Concerning unused principles (question 4) table 5.1 shows that all principles have been applied except the →4.7.3 *Liskov Substitution Principle* (LSP). The reason for that is that LSP deals with inheritance hierarchies and up to the development stage of the FeedReader experiment inheritance has not been used, yet. For that reason the principle is not removed.

It can furthermore be seen that KISS, MIMC and MP are by far the most utilized principles.

Listing 6.2: Possible Solutions for Design Decision #31

```
// Dependency Injection
public class Subscription
{
    ...
    public Subscription(Downloader downloader, Channel channel)
    {
        this.downloader = downloader;
        this.channel = channel;
    }
    ...
}

// Service Locator
public class ServiceLocator
{
    public Downloader getDownloader() { ... }
    public Channel getChannel() { ... }

    public void setDownloader { ... }
    public void setChannel { ... }
}

// Abstract Factory + Loadable Singleton
public abstract class Factory
{
    private Factory soleInstance;

    protected abstract Downloader getDownloaderDyn();
    protected abstract Channel getChannelDyn();

    public Downloader getDownloader()
    {
        return soleInstance.getDownloaderDyn();
    }
    public Channel getChannel()
    {
        return soleInstance.getChannelDyn();
    }

    public load(instance Factory)
    {
        this.soleInstance = instance;
    }
}
public ImplFactory extends Factory { ... }
public DummyFactory extends factory { ... }
```

Table 6.2: Utility of the Principle Language, Code Smells and SOLID

Design Decision	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13
PL	+	+	+	+	+	+	+	+	+	+	+	+	+
CS	+	+/-	+	+	+/-	-	-	+	+	-	-	+	+
SOLID	-	-	-	-	-	-	-	-	-	-	-	-	-

Design Decision	#14	#15	#16	#17	#18	#19	#20	#21	#22	#23	#24	#25
PL	+	+	+	+	+	+	+	+	+	+	+	+
CS	+	-	-	-	-	-	-	-	+	+	-	-
SOLID	+	-	-	+	-	-	-	-	-	-	-	-

Design Decision	#26	#27	#28	#29	#30	#31	#32	#33	#34	#35	#36	#37
PL	+	+	+	+	+	+	+	+	+	+	+	+
CS	+	-	-	+	+	-	-	-	-	+	-	-
SOLID	-	-	-	-	-	-	-	-	-	+	-	-

Design Decision	#38	#39	#40	#41	#42	#43	#44	#45	#46	#47	#48	#49
PL	+	+	+	+	+	+	+	+	+	+	+	+
CS	-	-	-	-	-	+	+/-	+	-	-	+/-	+
SOLID	-	-	-	+	-	+	+	-	-	-	+/-	-

This is because they are very central principles in the principle language as virtually all of the principles are somehow connected to them.

The last question addressed by the experiment is whether the approach helps better than the two competing approaches code smells and SOLID (question 5). Each of the 49 design decisions has been examined with respect to this question. Table 6.2 shows the result. For each Design decisions it lists a judgment whether the principle language (PL), code smells (CS) and SOLID provides guidance. + means that the approach helped, +/- means that the approach helped somehow but with undeniable limitations and - means that there is no guidance or the approach would prefer a suboptimal solution.

For all 49 design decisions the principle language provided guidance. The code smells approach helped in 13 cases (ca. 27 %) and provided limited help in further 4 cases (ca. 8 %). SOLID was only helpful in 5 cases (ca. 10 %) and provided minor help in one further case (ca. 2 %).

There is a wide range of design decisions and not all of them are judgments upon refactorings. Also prior to refactoring there are plenty of design decisions to take. Code smells are designed to guide refactoring hence they provide little guidance for the other tasks. The principle language is applicable in each of the steps described in section 5.1.5. This is the reason why it is applicable in much more situations. On the other hand code smells are much easier to apply as they are much more specific. Applying the principle language requires finding a characterizing set, rating the possibilities and judging the solutions whereas code

smells only require to match code against a fixed set of known to be bad structures.

The reason why SOLID is only applicable in such a small number of cases is, that not applicable to a large number of low-level design problems. The principle language provides guidance for all design problems beneath the architectural scale, ranging from decisions about modularization and module communication to more low-level tasks like the decisions about particular method signatures. SOLID is only about the higher-level design decisions and is thus applicable only in a small number of cases.

So while this only is a very small experiment the results indicate a significant benefit for the approach and the principle language. It is clearly more difficult to apply than code smells but it provides guidance in a large variety of situations while still being reasonably lightweight and fast to apply.

6.3 CoCoME

6.3.1 Setup

The Common Component Modeling Example (CoCoME) [68] was developed in 2006/2007 with the goal to define a common example for comparing component modeling techniques. There is a design documentation and a reference implementation. Besides the original implementation from 2007 there are several newer ones. The current reference implementation carries the date 2011. moreover there are several models using diverse component modeling techniques but these are not relevant for this experiment.

For this experiment the original design and the original reference implementations are examined. In the later implementations some design flaws may have been already corrected. So examining the original system will find the original design flaws. It furthermore gives the opportunity to compare the old version with the current one. Some flaws might have been removed which indicates maintenance because of these flaws.

CoCoME simulates a trading system as it can be found in supermarkets and similar stores. The trading system provides the software for the cash registers (scanning bar codes, handling payments, etc.), for servers in the sores (managing the stock, changing prices, etc.) and for a server central to the enterprise (managing stores, creating reports, etc.).

The system was developed in a traditional plan-driven manner and was designed and implemented by researchers. It uses Java RMI, JPA/Hibernate, and JMS/Apache ActiveMQ. With about 9000 (original reference implementation) and about 19000 lines of code (2011 version) the system is still reasonably small but far from being a toy example. It is furthermore a distributed system using several middleware technologies which creates design problems similar to those in “real” enterprise systems.

For the experiment the original system is reviewed, design flaws are documented as well as the characterizing set of principles revealing the flaws. Additionally it is noted whether the new version of the system still contains the design flaw. Note that only design flaws are recorded which are not architectural flaws. Also coding faults (concurrency problems, bad exception handling, etc.) and anything else below the design level is out of scope. During the review such defects were found but not documented.

6.3.2 Questions

1. Are there design flaws in CoCoME which could have been prevented by using the principle language? This means: Are there design flaws that can be explained and described using the principle language so a proper usage of it would have led to a better system? If this is the case, this will be a sign that the principle language is helpful and its usage prevents design flaws.

2. Are these design flaws removed in the newer version of the system? This would indicate that over time the flaw was detected and removed during maintenance. If the flaw had not been present in the first place, this rework would not have been necessary. So in this case this would be an indication that the application of the principle language can reduce maintenance effort.

6.3.3 Results

Appendix B shows the protocol of the experiment. The review yielded 51 design flaws (plus several repetitions of the same faults). The vast majority of these flaws are not very critical. The design defect which occurred the most was a misleading or otherwise suboptimal identifier. But there are also a few more problematic flaws.

The most problematic one probably is design flaw #5: The data layer of the inventory subsystem is realized as a separate component `Data` which is implemented as a class `DataImpl` having an `interface` `DataIf`. The sole purpose of the class is to provide access methods for three subcomponents. The class has no clear advantage and is also not prescribed by the specification[2]. `DataImpl` is instantiated by a class named `DataIfFactory`. The code of this class is shown in listing 6.3.

Listing 6.3: The DataIfFactory Class

```
public class DataIfFactory {
    private static DataIf dataaccess = null;
    private DataIfFactory() {}

    public static DataIf getInstance() {
        if (dataaccess == null) {
            dataaccess = new DataImpl();
        }
        return dataaccess;
    }
}
```

Essentially `DataIfFactory` resembles a mixture between the design patterns *factory* and *singleton* [3]. The latter one is important here. The purpose of a singleton is to make a single instance of a class globally accessible. Here `DataImpl` is not ensured to be only instantiated

2 this is design flaw #7

once as it still has a public constructor. Nevertheless the "factory" class makes it globally accessible. In every part of the software `DataIfFactory.getInstance()` can be used to get hold of the data component. And since `DataIf` makes the three subcomponents accessible, also these are accessible from everywhere. There is no need to pass a reference around.

The problem with this approach is, that this creates a tight coupling between every class that uses the data component and the concrete class `DataImpl` respectively all concrete implementations of the subcomponents. There is no way to reuse, or test any dependent class without also keeping the data component or changing the code. The →4.5.2 *Low Coupling* (LC) Principle argues against this. The →4.3.6 *Rule of Explicitness* (RoE) is also against this factory singleton. It obfuscates the real dependencies as it is not clear from the interface of the dependent classes that they depend on the data component.

Design flaw #20 lists roughly the same problem for another major part of CoCoME: There is a class named `ApplicationFactory` which has the same problems. So apart from the database management system, two out of thee main components or six out of eight subcomponents of the inventory subsystem are not directly reusable and there is not even the possibility to test these components in isolation as they are always integrated. There is no way to use dummy components in order to ensure test isolation.

This also goes against the component idea CoCoME especially was designed for. A component shall be independent from the rest of the system. It shall be independently reusable, independently testable, and independently substitutable. While substitutability is not hampered like the other two goals, it is clear that making the components more or less singletons is a severe design flaw which is not consistent with the idea of component-based software engineering.

A much better solution would be to use dependency injection [70]. This would mean, instead of making the components globally available, references to them would have to be supplied to the dependent components for example via constructor parameters, setter methods. Factories also play a major role in this approach but they work differently. Listing 6.4 shows how such a solution can look like.

Listing 6.4: A Factory for Dependency Injection

```
public class StoreServerFactory
{
    private StoreServerFactory() {}

    public static StoreIf createStoreComponent()
    {
        return new StoreImpl(createPersistenceComponent(),
            createStoreQueryComponent);
    }

    // these are the tree subcomponents of the Data Component
    public static PersistenceIf createPersistenceComponent() {
        return new PersistenceImpl(); }
    public static StoreQueryIf createStoreQueryComponent() { return
        new StoreQueryImpl(); }
```

Table 6.3: Design Flaws in CoCoME Which Have Been Fixed

Flaw	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14
Fixed				✓						(?)	(✓)			
Flaw	#15	#16	#17	#18	#19	#20	#21	#22	#23	#24	#25	#26	#27	
Fixed			(?)	✓	✓	✓	✓		(?)	✓				
Flaw	#28	#29	#30	#31	#32	#33	#34	#35	#36	#37	#38	#39	#40	
Fixed	✓		✓											
Flaw	#41	#42	#43	#44	#45	#46	#47	#48	#49	#50	#51			
Fixed	✓	✓	✓	✓	✓	✓	(?)	✓	✓	✓	✓			

```
    public static EnterpriseQueryIf createEnterpriseQueryComponent()
        { return new EnterpriseQueryImpl(); }
}
```

An explicit class for the data component is not necessary. Rather the subcomponents are passed directly to the dependent components without using the indirection created by the current realization of the Data component. This is consistent with the specification given in [68]. The code in the listing is a bit simplified. In the real system there are further dependencies which have to be taken into account. The code just shows the principle.

Injecting the dependencies has the advantage that coupling stays low and there is no direct dependency on the concrete implementation (LC). This makes the components reusable and testable. Furthermore the dependencies are directly visible from the interface of the component (RoE). And additionally the components are more general as arbitrary implementations for the dependencies can be supplied to the components (see →4.3.5 *Generalization Principle* (GP)). So {LC, RoE, GP} is the characterizing set of principles which would have helped avoiding this design flaw[3].

For all 51 design flaws characterizing sets could be found. So question 1 can be answered positively: There are flaws in CoCoME which can be explained using the principle language. It can be assumed that a consequent usage of the principle language would have prevented many of the flaws found in the system.

Question2 can also be answered based on the given data. Table 6.3 shows which flaws have been corrected which is assumed to be correlated with the maintenance work on the flaws. ✓ means the flaw has been fixed, (✓) means the flaw has not been fixed but recognized as such, and (?) means the flaw is not directly fixed but the design is different now.

There are 51 flaws and 18 of them have been fixed, one has been recognized and in another four cases the design was changed in a different way. So a large part of the design flaws showed up during the development and caused maintenance. If the flaws had not been make in the first place, less maintenance work would have been necessary. This indicates that the

3 Note that basically the same question also arose in the FeedReader example. See section 6.2

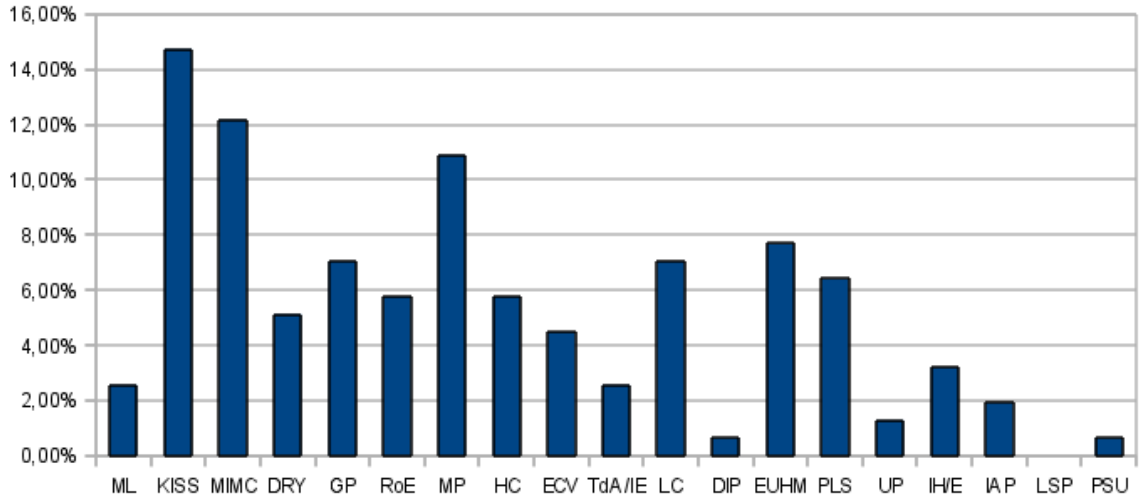


Figure 6.1: Distribution of the Principle Usage in the FeedReader Example

principle language can help reducing maintenance effort (question 2).

Apart from the discussed questions two other observations have been made. First of all it is noticeable that the distribution of the principles used in the characterizing sets is completely different. Figure 6.1 shows the distribution for the FeedReader example and figure 6.2 shows it for the CoCoME example. In the first one KISS and MIMC were used most whereas in the latter MP and PLS were much more used than the other ones. The reason for that is that the →4.4.1 *Model Principle* (MP) and the →4.6.2 *Principle of Least Surprise* (PLS) are used to reason about identifier naming. A module identifier should explain the module in such a way that its behavior does not surprise the user and it should be named according to some model. As suboptimal identifiers were the most common design flaw, these two principles are used most. Furthermore the task was completely different. In the CoCoME example only design flaws have been recorded whereas in the FeedReader example every design decision was examined.

The second observation is the usage of the →4.4.1 *Model Principle* (MP). MP was used very often and in a large variety of cases. It is a broad principle which has many aspects. MP is about identifier naming (flaws #3, #4, #9, #14 and many others), domain modeling (flaw #12), the usage of objects (#16, #27, #31, ...), the assignment of responsibilities (#18, #25, ...), the natural relationships among objects (#43, #45) and other aspects. This variety makes MP sometimes more difficult to apply than other principles. A future improvement of the principle language could therefore be to add specializations of the model principle and in return for that maybe replace some of the less used principles by fewer more general ones.

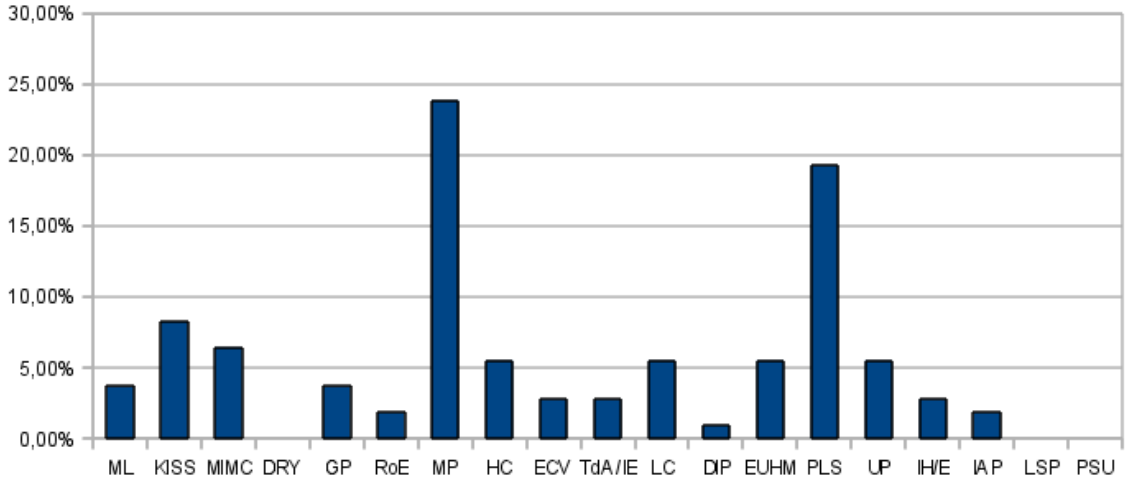


Figure 6.2: Distribution of the Principle Usage in the CoCoME Example

6.4 Conclusion

The results of the two experiments look promising. They showed that the principle language is capable of producing characterizing sets for arbitrary design problems. The principle language approach also proved to be applicable for a larger set of problems than Martin Fowler's code smells and Robert C. Martin's SOLID principles. Despite being some helpful guidance the results indicate that a consequent application of the principle language approach may be beneficial for maintainability of the software to develop.

But of course the experiments also have their limitations. The examined software is rather small (especially the first one), the examined systems have been designed and developed by researchers (and not practitioners) and although there was some kind of maintenance for the CoCoME system, this is certainly different from a system which is maintained because it is actually used. Moreover none of the experiments involved communication about design decisions or any other people except the author of the principle language applying it.

So this can just be the start. More experimentation is necessary in order to gain confidence in the approach and the language. It is especially worthwhile to examine the utility of the principle language with respect to different levels of experience or different processes.

7 Outlook and Further Work

Principle languages are proposed in this thesis. This is a whole new research area so it is naturally under-explored. Many open questions remain and many possibilities for further research exist. As already stated in section 5.2 surprisingly few principles have been subject to scientific assessment. While many of them are already accepted in practice, research has not caught up on that. Research examining the validity of single principles is still necessary.

Further examinations of the principle language proposed here may also be worthwhile. Two small experiments have been presented in chapter 6. They support the principle language but further evidence is still needed. Larger studies may improve confidence in the approach and in the language. The usefulness of the principle language may also be assessed with respect to certain domains (information systems, embedded systems, etc.) and different experience levels of designers (students, practitioners, highly experienced designers, etc.). Also its value for teaching could be examined.

Research might also result in an improvement of the principle language. Some principles might be missing, others might be dispensable. Even more the navigation relationships may change. Section 6.3 already hints some possible improvements. In particular future versions of the principle language may include specializations for the →4.4.1 *Model Principle* (MP).

This thesis proposes a principle language for object-oriented design. Further principle languages may follow. First of all there may be principle languages for other forms of software design like user interface design, framework design, database schema design, communication protocol design, or mobile application design. There may also be principle languages tailored for specific domains: for the design of control systems, enterprise applications, games, multimedia systems, aerospace applications, or software for the the automotive industry. It is still unclear whether such a tailoring is possible or helpful. The goal of this thesis is just to provide a general-purpose principle language. Tailorability is still an open question.

Other possibilities for principle languages include those specifically concerned with certain non-functional requirements like performance, reliability, or security. Furthermore principle languages on higher or lower levels of abstraction may be envisioned: principle languages for architecture, requirements analysis, algorithm design or coding. As explained in the section 2.1.2 a principle language for low-level design as it is presented here might be most promising but research may show that principle languages are also helpful on other levels of abstraction. This is also still an open research question.

And lastly there are also other programing paradigms. This principle language is specifically constructed for guiding in object-oriented design. But there is also procedural programming (C, Pascal, ...) and functional programming (LISP, Haskell, ...), there are fourth generation programming languages (Progress, ABAP, ...) each of which have unique requirements in how to design systems developed using these paradigms. And although partly also

object-oriented, scripting languages may benefit from an own principle language, too.

Another question which is still unexplored is the precise relationship between principles and patterns and anti-patterns. The relationship between the idea of principles and the idea of patterns is described in section 2.3.1. But it is still an open question how single patterns and single principles relate, which types of relationships there are, and if this relationship can be exploited in some way.

So there is plenty of research potential, many open questions and many possibilities for additional principle languages.

8 Conclusion

Software design is a complex task requiring knowledge, skill and experience. But while knowledge can be taught, gaining skill and experience takes time. But it is not only difficult to do design and to make sound design decisions. Communicating the reasons for design choices is difficult as well. Experienced designers may be tempted to just refer to their "experience" when discussing designs. But doing so is not a convincing reason for a design choice.

This thesis addressed these problems by examining software design principles. These principles are memorable, informal design guidelines which distinguish good solutions from bad solutions with respect to a specific aspect. Seasoned designers and well-known researchers formulated these principles making their tacit design experience teachable and learnable. While in the past such principles have mainly been discussed in isolation, in this work they have been related to each other. Just like patterns have been interconnected forming pattern languages, this work interconnected principles in order to create a principle language. The result is part of this thesis and has also been documented using a wiki, which will go online shortly.

Along with the principle language an analytic design approach has been developed and described. This approach is light-weight, makes use of the proposed principle language and helps making arbitrary low-level design decisions. In plan-driven development projects it is used during the design and coding phase and in agile processes it fits nicely with test-driven development.

While constructing the principle language it has been recognized that the relationships between the principles are central to the approach but non-trivial. The key finding here is that the relationships in the principle language have to be designed in a way that facilitates principle discovery. The important aspect about the relationships is how they are used.

Two experiments have been conducted in order to evaluate the approach as well as the principle language. The first experiment examines the utility of the approach compared to Martin Fowler's code smells and Robert C. Martin's SOLID principles. This experiment showed that the principle language is applicable in more design situations. The second experiment indicated that a consequent usage of the principle language may have a positive effect on software quality. Although this is just a first evidence, the results look promising and suggest to place further effort into the topic. Principle languages have been newly proposed in this thesis and the area comprises plenty of possibilities for further research. In particular there is a lack in evidence for several principles, studying the utility of the approach for different tasks and people is worthwhile and there are several possibilities for constructing principle languages for other contexts. So it is is suggested to continue research in the area of principle languages. This thesis can only be a start.

Bibliography

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language: Towns, Buildings, Construction.* New York: Oxford University Press, 1977. [Online]. Available: https://opac.ub.uni-kl.de/F/?func=find-c&ccl_term=idn%3DTT001547422&local_base=KLU01

[2] K. Beck and W. Cunningham, "Using pattern languages for object-oriented programs," http://c2.com/doc/oopsla87.html, Computer Research Laboratory, Tektronix, Inc., Tech. Rep. CR-87-43, September 1987.

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

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996, vol. 1: A System of Patterns.

[5] F. P. Brooks, "No silver bullet: Essence and accidents of software engineering," *IEEE Computer*, vol. 20, pp. 10–19, 1987.

[6] L. Bass, P. Clemens, and R. Kazman, *Software Architecture in Practice*, 2nd ed., ser. SEI Series in Software Engineering. Addison-Wesley, 2003.

[7] P. Sommerlad, "Design patterns are bad for software design," *IEEE Software*, vol. July/August, pp. 68–70, 2007.

[8] M. Fowler, *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[9] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2003.

[10] ——, *UML for Java Programmers.* Prentice Hall International, 2003.

[11] ——, "The principles of ood," http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

[12] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, 2004.

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

[14] B. Meyer, *Object-Oriented Software Construction*, second edition ed. Prentice Hall, 1997.

[15] E. S. Raymond, *The Art of UNIX Programming.* Addison-Wesley, 2003, http://www.catb.org/~esr/writings/taoup/html/.

[16] A. M. Davis, *201 Principles of Software Development*. McGraw-Hill, 1995.

[17] A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering*. Addison-Wesley, 2003.

[18] Anonymous, "Wiki history," in *Portland Pattern Repository*, October 8 2012.

[19] M. Jackson, *Software Requirements and Specifications: A Lexicon of Software Practice, Principles and Prejudices*. ACM Press, 1995.

[20] N. T. Spark, "The fastest man on earth," *Annals of Improbable Research*, vol. 9, pp. 4–26, 2003.

[21] E. S. E. Raymond, "Jargon file: Murphy's law," http://www.catb.org/jargon/html/M/Murphys-Law.html.

[22] "Java api: String.replacefirst," http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#replaceFirst(java.lang.String,java.lang.String).

[23] "Java api: Pattern," http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html.

[24] "Java api: Date," http://docs.oracle.com/javase/7/docs/api/java/util/Date.html.

[25] "Java api: Calendar," http://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html.

[26] S. Colebourne, "Jep 150: Date & time api," http://openjdk.java.net/jeps/150.

[27] B. R. Rich, *Clarence Leonard (Kelly) Johnson 1910—1990: A Biographical Memoir*. Washington DC: National Academies Press, 1995, http://www.nap.edu/html/biomems/cjohnson.pdf.

[28] C. A. R. Hoare, "The emperor's old clothes / the 1980 acm turing award lecture," http://awards.acm.org/images/awards/140/articles/4622167.pdf, 1980.

[29] E. S. E. Raymond, "Jargon file: Kiss principle," http://www.catb.org/jargon/html/K/KISS-Principle.html.

[30] V. R. Gibson and J. A. Senn, "System structure and software maintenance performance," *Commun. ACM*, vol. 32, no. 3, pp. 347–358, Mar. 1989. [Online]. Available: http://doi.acm.org/10.1145/62065.62073

[31] C. F. Kemerer, "Software complexity and software maintenance: A survey of empirical research," *Annals of Software Engineering*, vol. 1, pp. 1–22, 1995.

[32] B. Boehm, "Software engineering economics," *Software Engineering, IEEE Transactions on*, vol. SE-10, no. 1, pp. 4–21, 1984.

[33] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *Psychological Review*, vol. 101, pp. 343–352, 1955.

[34] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Communications of the ACM*, vol. 27, pp. 42–52, 1984.

[35] K. E. Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S. N. Rai, "The optimal class size for object-oriented software," *Software Engineering, IEEE Transactions on*, vol. 28, no. 5, pp. 494–509, 2002.

[36] J. Rosenberg, "Some misconceptions about lines of code," in *Software Metrics Symposium, 1997. Proceedings., Fourth International*, 1997, pp. 137–142.

[37] N. Fenton and M. Neil, "A critique of software defect prediction models," *Software Engineering, IEEE Transactions on*, vol. 25, no. 5, pp. 675–689, 1999.

[38] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "An empirical study evaluating depth of inheritance on the maintainability of object-oriented software," *Empirical Software Engineering, An international Journal*, vol. 1, pp. 109–132, 1996.

[39] B. Unger and L. Prechelt, "The impact of inheritance depth on maintenance tasks," Universität Karlsruhe, Tech. Rep., 1998.

[40] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's University at Kingston, Tech. Rep., 2007.

[41] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 485–495.

[42] D. L. Parnas, "Designing software for ease of extension and contraction," *Software Engineering, IEEE Transactions on*, vol. SE-5, no. 2, pp. 128–138, 1979.

[43] A. J. Perlis, "Epigrams on programming," *SIgPLAN Notices*, vol. 17, no. 9, pp. 7–13, 1982.

[44] T. Peters, "The zen of python," http://www.python.org/dev/peps/pep-0020/.

[45] G. van Rossum, "Python's design philosophy," http://python-history.blogspot.de/2009/01/pythons-design-philosophy.html.

[46] M. Fowler, "To be explicit," *IEEE Software*, vol. November/Devember, pp. 10–15, 2001.

[47] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language*, 1993.

[48] G. Booch, *Object-Oriented Design With Applications*. The Benjamin/Cummings Publishing Company, 1991.

[49] L. Chou, "Lambda the ultimate: Why are objects so unintuitive?" http://lambda-the-ultimate.org/node/3265, 2009.

[50] B. Jacobs, "Oop and "modeling the real world"," http://www.geocities.com/tablizer/model.htm.

[51] Anonymous, "Oop not for domain modeling," in *Portland Pattern Repository*, January 10 2013.

[52] W. P. Stevens, G. J. Myers, and L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[53] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[54] P. Coad, *Object Models: Strategies, Patterns and Applications.* Prentice Hall, 1996.

[55] A. Hunt and D. Thomas, "The art of enbugging," *IEEE Software*, vol. 20, pp. 10–11, Jan/Feb 2003, http://www.ccs.neu.edu/research/demeter/related-work/pragmatic-programmer/jan_03_enbug.pdf.

[56] G. J. Myers, *Reliable Software through Composite Design.* Mason and Lipscomb Publishers, 1974.

[57] M. Page-Jones, *The Practical Guide to Structured Systems Design.* YOURDON Press, 1980.

[58] R. C. Martin, "Object oriented design quality metrics: An analysis of dependencies," 1994.

[59] G. James, "The tao of programming," http://www.canonical.org/~kragen/tao-of-programming.html.

[60] "Msdn: Queue class," http://msdn.microsoft.com/en-us/library/system.collections.queue(v=vs.110).aspx.

[61] "Delphi api documentation: System.contnrs.tqueue methods," http://docwiki.embarcadero.com/Libraries/XE3/en/System.Contnrs.TQueue_Methods.

[62] P. Rogers, "Encapsulation is not information hiding," http://www.javaworld.com/javaworld/jw-05-2001/jw-0518-encapsulation.html, 2001.

[63] A. Poetzsch-Heffter, *Konzepte objektorientierter Programmierung.* Springer Verlag, 2009.

[64] "Delphi api documentation: System.tdatetime," http://docwiki.embarcadero.com/Libraries/XE3/en/System.TDateTime.

[65] B. Liskov, "Data abstraction and hierarchy," in *ACM Sigplan Notices*, vol. 23, no. 5. ACM, 1987, pp. 17–34.

[66] B. H. Liskov and J. M. Wing, "Behavioral subtyping using invariants and constraints," Tech. Rep., 1999.

[67] R. C. Martin, "The bowling game kata," http://butunclebob.com/ArticleS.UncleBob.TheBowlingGameKata.

[68] A. Rausch, R. Reussner, R. Mirandola, and F. Plášil, Eds., *The Common Component Modeling Example.* Springer Verlag, 2008.

[69] R. A. Board, "Rss 2.0 specification," http://www.rssboard.org/rss-specification.

[70] M. Fowler, "Inversion of control containers and the dependency injection pattern," http://martinfowler.com/articles/injection.html, 2004.

[71] C. Rehn, "Singletons nach fowler," http://www.christian-rehn.de/2009/12/06/singletons-nach-fowler/, 2009.

[72] "Springsource.org," http://www.springsource.org/.

Date of access to the referenced URLs: April 14, 2013

List of Figures

List of Tables

List of Listings

A Protocol of the Design Decisions in FeedReader

This is a protocol of the design decisions which were taken while developing the FeedReader example. For each design decision, a brief description of the problem is given, the result of applying the principle language (PL), Fowler's code smells (CS), and Martin's SOLID principles, as well as the result of the decision.

For the principle language the path taken through the language is given. The line lists a subgraph of the principle language in the following notation:

- `A -> B` means `A` is considered and it lists `B` as a related principle. `B` qualifies and is also added to the characterizing set.
- `A => B` means `A` is considered and it lists `B` as a generalization or specialization. `A` is then replaced by `B`.
- Several steps in the principle discovery are separated by `|`.
- If the line lists principles which have no other principles leading to them, they are starting principles.
- `A -> B -> C` is a shorthand for `A -> B | B -> C`
- `A -> B, C` is a shorthand for `A -> B | A -> C`
- `PLS => EUHM -> KISS, MP | RoE -> ML => EUHM` means that the starting principles were PLS and RoE. PLS was replaced by EUHM, EUHM lead to KISS and MP. Roe lead to ML which was also replaced by EUHM. So the characterizing set is {EUHM, KISS, MP, RoE}.

For the code smells the protocol lists the smell, an arrow (`-->`) and the refactoring the smell initiated.

The SOLID section just lists the principle which is applicable.

Furthermore each of the three approaches gets a rating. `(+)` means that the approach helped, `(+/-)` means that the approach helped somehow but had some limitations and `(-)` means that the approach did not help. If the approach was not applicable (`--`) the rating is obviously `(-)`.

```
-> also consider
=> replaced by
--> leads to refactoring
```

```
================================================================================

US1: I want to read an RSS feed in order to be informed.

#1
Problem: Parameter for PLFeeds.downloadFeed: URL or String
PL: PLS => EUHM -> KISS, MP | RoE -> ML => EUHM  (+)
CS: Primitive Obsession --> replace data value with object (+)
SOLID: --
Result: Use URL

#2
Problem: Procedural static or OO main class
PL: MP -> KISS (+)
CS: ? --> Convert procedural design to objects (+/-)
SOLID: --
Result: OO

#3
Problem: Duplicate code for running the dummy web server in PLFeedTest
PL: DRY -> KISS => MIMC (+)
CS: duplicated code --> extract method (+)
SOLID: --
Result: runDummyWebserver()


--------------------------------------------------------------------------------

US2: I want the output to be readable so I don't have to read ugly XML.

#4
Problem: Duplicate code for sample RSS in PLFeedTest
PL: DRY -> KISS => MIMC (+)
CS: duplicated code --> extreact method (+)
SOLID: --
Result: createDummyRSS()

#5
Problem: Method parameters for createDummyRSS() are all string
PL: MIMC | ML -> KISS -> MP (+)
CS: long parameter list --> ?
| ? --> replace method with method object (+/-)
SOLID: --
```

```
Result: Leave it a method for now

#6
Problem: Parse XML or use RSS library
PL: KISS -> GP, MIMC (+)
CS: --
SOLID: --
Result: normally 3rd party library; here parsing XML anyway

#7
Problem: How to parse XML: DOM, SAX, StAX, ...
PL: KISS -> GP, MIMC (+)
CS: --
SOLID: --
Result: DOM+XPath
Note: Speculative Design probably would have resulted in using SAX because
maybe in the future DOM will be too slow.

#8
Problem: Long method prettifyFeed()
PL: MIMC -> MIMC (+)
CS: long method --> extract method (+)
SOLID: --
Result: parseFeedItem() and parseFeedMetaData()

#9
Problem: parseFeedItem() has parameters items and i
PL: EUHM -> IAP, MP | GP -> RoE -> MIMC (+)
CS: Long parameter list(?), primitive obsession
--> introduce parameter object (+)
SOLID: --
Result: replace parameters items and i by item

#10
Problem: Inconsistent parameter lists: parseFeedItem(StringBuilder result,
XPath xpath, Node item) and parseFeedMetaData(StringBuilder result,
Document document, XPath xpath)
PL: UP (+)
CS: --
SOLID: --
Result: reordered parameters of parseFeedItem() and parseFeedMetaData()

#11
```

```
Problem: Should parseFeedItem() and parseFeedMetaData() throw
XPathExpressionException or rather catch and retrow as RuntimeException?
PL: KISS, EUHM (+)
CS: --
SOLID: --
Result: throw XPathExpressionException // methods are private,
so KISS is more important


--------------------------------------------------------------------------

US3: I want to have a nice graphical UI so I don't have to use the console.

#12
Problem: feedReader.prettifyFeed(feedReader.downloadFeed(...)) is ugly
PL: IE -> MP, IH/E (+)
CS: feature envy --> extract method + move method (+)
SOLID: --
Result: processURL()

--------------------------------------------------------------------------

US4: I want the GUI to have a table for the feed items so they are
clearly arranged.

#13
Problem: FeedItem.getLink(): String or URL?
PL: PLS => EUHM -> KISS, MP | RoE -> ML => EUHM  (+)
CS: Primitive Obsession --> replace data value with object (+)
SOLID: --
Result: Use URL

#14
Problem: PLFeeds is an artificial construct
PL: HC -> LC -> MP | HC -> MIMC, ECV (+)
CS: large class --> extract method (+)
SOLID: SRP (+)
Result: FeedParser

#15
Problem: FeedParser class is still artificial
PL: MP | EUHM => PLS (+)
CS: --
```

```
SOLID: --
Result: FeedParser is Channel now and takes the URL in the constructor

#16
Problem: There is a temporal dependency between Channel.getTitle()
and Channel.processURL()
PL: PLS -> EUHM | LC (+)
CS: --
SOLID: --
Result: lazy loading using the method ensureFeedParsed();

#17
Problem: Now getTitle() and getDescription() may throw
ClientProtocolExceptions, etc.
PL: PLS | HC -> ECV -> MP | HC -> LC, MIMC (+)
CS: --
SOLID: SRP (+)
Result: Downloader class

#18
Problem: Should Channel get a Downloader object or rather the
downloaded feed as string?
PL: KISS -> GP | LC (+)
CS: --
SOLID: --
Result: String

#19
Problem: Should Channel do the parsing in the constructor or later
in a separate method?
PL: PLS, EUHM (+)
CS: --
SOLID: --
Result: later in a separate method

#20
Problem: Channel.processURL() does not process any URL
but returns the feedItems
PL: PLS -> MP (+)
CS: --
SOLID: --
Result: processURL is getFeedItems() now
```

```
#21
Problem: Again there is a temporal dependency between Channel.parseFeed()
and Channel.getTitle(), etc.; values are null before the feed is parsed
PL: PLS -> MP -> KISS (+)
CS: --
SOLID: --
Result: initialize with empty string; this does not completely solve the
problem but now the behavior is clear by the model

#22
Problem: duplicated code for presenting error message
PL: DRY -> KISS (+)
CS: duplicated code --> extract method (+)
SOLID: --
Result: showError()

-------------------------------------------------------------------------

US5: I want the GUI to show the description of each item so I can
better decide which link to follow.

#23
Problem: duplicated creation of FeedItems in FeedItemTest
PL: DRY -> KISS (+)
CS: duplicated code --> extract method (+)
SOLID: --
Result: fixture setUp()

-------------------------------------------------------------------------

US6: I want to be able to subscribe to feeds so I don't have to type the
URL again and again.

#24
Problem: Make SubscriptionDialog modal (and add a showModal() method) oder
modeless (and add a listener)?
PL: LC -> KISS -> MIMC (+)
CS: --
SOLID: --
Result: modal
Note: temporal coupling

#25
```

```
Problem: Return type of showModal(): bool, int or own enum?
PL: PLS -> UP -> ML => EUHM (+)
CS: --
SOLID: --
Result: enum

#26
Problem: duplicated code for closing dialog
PL: DRY -> KISS (+)
CS: duplicated code --> extract method (+)
SOLID: --
Result: close()

#27
Problem: Let SubscriptionDialog change subscriptions directly
or just provide strings?
PL: TdA -> MP, LC, IH/E | MP -> ECV | IH/E -> EUHM (+)
CS: --
SOLID: --
Result: change directly

#28
Problem: Use DefaultListModel or create an own one?
PL: KISS -> MP, GP | MP -> HC (+)
CS: --
SOLID: --
Result: DefaultListModel

#29
Problem: Code duplication in SubscriptionTest:
constructing Subscription object
PL: DRY -> KISS (+)
CS: duplicated code --> extract method (+)
SOLID: --
Result: fixture

#30
Problem: add Subscription.getChannel() or forward requests to it?
PL: TdA -> IH/E, MIMC (+)
CS: message chains --> hide delegate (+)
SOLID: --
Result: forward requests
```

```
#31
Problem: How to make Subscription testable? It needs a Downloader
and a Channel but these should be stubbed away for  the purpose
of test isolation.
PL: LC -> KISS, RoE | KISS -> ML, GP (+)
CS: --
SOLID: --
Result: dependency injection
Note: Alternatives: dependency injection, abstract factory
as a loadable singleton, service locator

#32
Problem: Use a DI framework or inject by hand?
PL: MIMC -> KISS -> GP -> RoE (+)
CS: --
SOLID: --
Result: by hand

#33
Problem: Wich injection type shall be used?
PL: KISS -> ML => EUHM (+)
CS: --
SOLID: --
Result: constructor injection
Note: Alternatives: constructor injection, interface injection,
setter injection

#34
Problem: Shall the dependencies be injected directly or shall
a factory be injected?
PL: MIMC | RoE (+)
CS: --
SOLID: --
Result: inject depedndencies directly
Note: Alternatives: inject depedndencies or inject a factory

#35
Problem: Which class shall inject the dependencies? A class of
the higher layer or a factory?
PL: HC -> ECV, LC, MIMC (+)
CS: divergent change --> extract class (+)
SOLID: SRP (+)
Result: use a factory as injector
```

```
#36
Problem: Which kind of factory shall be used? Concrete factory
or abstract factory?
PL: DIP -> MIMC (+)
CS: --
SOLID: DIP (-)
Result: concrete factory
Note: SOLID would propose a more complicated solution.

#37
Problem: Configure factory by code or config file?
PL: KISS -> GP -> RoE | KISS -> ML (+)
CS: --
SOLID: --
Result: code

#38
Problem: Make FeedReaderFactory a utility class?
PL: KISS -> GP (+)
CS: --
SOLID: --
Result: static

#39
Problem: Parameters of Subscription constructor?
Only dependencies or also URL?
PL: KISS => MIMC | ML => EUHM -> IAP -> LC (+)
CS: long parameter list (-)
SOLID: --
Result: Dependencies and URL
Note: CS would only consider the length of the parameter list.

#40
Problem: SubscriptionDialog currently needs an empty Subscription
in order to fill it. But empty subscriptions are not allowed
anymore since #39
PL: MP -> KISS, ECV | ECV -> LC -> HC (+)
CS: --
SOLID: --
Result: Let the SubscriptionDialog create Subscriptions.
Note: Alternatives: let the dialog create subscriptions or pass in a
Subscription with dummy values or make the Dialog completely unaware
```

```
of Subscriptions and have it just provide strings.

#41
Problem: Make dialog aware of the task carried out (add or edit a
subscription) or allow creating a SubscriptionDialog with a
null as Subscription?
PL: MP -> HC (+)
CS: --
SOLID: SRP (+)
Result: Let the SubscriptionDialog create an own subscription
if none is supplied.

#42
Problem: Add a method getSubscription() or let showModal() return
a Subscription?
PL: PLS -> EUHM -> IAP (+)
CS: --
SOLID: --
Result: let showModal() return a Subscription

#43
Problem: prettifyFeed() builds up header and feed items
PL: HC -> PSU, MIMC, ECV (+)
CS: long method --> extract method (+)
SOLID: SRP (+)
Result: extract method buildUpHeader

#44
Problem: Let the extracted method print the header
or just return a string?
PL: GP | HC (+)
CS: divergent change (+/-)
SOLID: SRP (+)
Result: don't print

#45
Problem: Channel.getDescription() needs to be accessible to the clients
of Subscription. Make channel visible to the outside (getChannel())
or mirror the method (Subscription.getDescription() delegating the call)?
PL: TdA -> IH/E, MIMC, MP (+)
CS: message chains --> hide delegate (+)
SOLID: --
Result: mirror the description of the channel in Subscription
```

```
#46
Problem: Supply the Subscription parameter to the SubscriptionDialog
in constructor or showModal()?
PL: PLS | GP -> RoE (+)
CS: --
SOLID: --
Result: showModal()

#47
Problem: Overload showModal or pass in null explcitly?
PL: PLS | RoE (+)
CS: --
SOLID: --
Result: overload

#48
Problem: For setting the column width let the table model access the
column model of the table, create a method in MainFrame instead
or use a dedicated ColumnModel object?
PL: MP -> KISS, HC, ECV, LC | HC -> MIMC | LC -> IH/E (+)
CS: inappropriate intimacy --> ? (+/-)
SOLID: SRP (+/-)
Result: Let the table model do that for now. It's simpler.

#49
Problem: runDummyWebserver is duplicated
PL: DRY -> KISS (+)
CS: duplicated code --> extract class (+)
SOLID: --
Result: DummyWebServer class
```

B Protocol of the Design Flaws in CoCoME

This is a protocol of the design review of CoCoME. All the design flaws which have been discovered are listed. For each flaw a rough description is given, the way through the principle language in the same notation as in appendix A and a remark whether the flaw has been fixed in the new version of CoCoME.

```
#1
Problem: StoreWithEnterpriseTO inherits from StoreTO; not specified
PL: MIMC -> KISS | UP
New version: still present

#2
Problem: ComplexOrderTO inherits from OrderTO and OrderTO is never used
PL: MIMC -> KISS | UP
New version: still present
Recurring: ProductWithSupplierTO
Note: Removing this unnecessary inheritance would simplify the TOs a lot
although there is not much complexity inthem anyway.

#3
Problem: ProductWithStockItem should be StockItem
PL: MP
New version: still present

#4
Problem: ScannerController sould be named Scanner; in OO the objects are
representatives
PL: MP
New version: solved
Recurring: all the other Controllers

#5
Problem: DataIfFactory makes the data component globally visible
PL: LC -> RoE -> GP
New version: still present

#6
```

```
Problem: DataIfFactory.getInstance() returns DataIf not DataIfFactory
PL: PLS -> UP
New version: still present

#7
Problem: DataIf has no advantage and is also not prescribed by the
specification
PL: MIMC
New version: still present

#8
Problem: DataIf.getPersistenceManager() ends with -Manager the other
methods with If
PL: UP -> PLS
New version: still present

#9
Problem: TransactionContext should be named Transaction
PL: MP
New version: still present

#10
Problem: TransactionContext is a thin wrapper around
javax.persistence.EntityTransaction which itself is standard with
several implementations; it adds nothing
PL: KISS -> MIMC, GP
New version: in the new version there is TransactionWrapper which has
some value
Recurring: PersistenceContext

#11
Problem: PersistenceContext which is a thin wrapper around EntityManager
is casted to PersistenceContextImpl in order to reveal the
EntityManager
PL: IH/E | ML -> KISS
New version: still present but the problem has been recognized
(there is a comment)

#12
Problem: the domain model is anemic;
see http://www.martinfowler.com/bliki/AnemicDomainModel.html
PL: MP | TdA/IE -> HC
New version: still present
```

```
#13
Problem: StoreQueryIf.queryProducts(): unclear which products;
better identifier: getAllProductsInStrore()
PL: PLS
New version: still present

#14
Problem: queryStockItem: bad identifier; not the items are queried
but they are queried for
PL: PLS -> MP
New version: still present
Recurring: all query methods

#15
Problem: queryStockItem should be named getStockItemByBarCode
PL: PLS -> MP
New version: still present

#16
Problem: queryStockItem gets a soreId instead of a store
PL: MP
New version: still present

#17
Problem: ProductDispatcher not named -Impl
PL: UP -> PLS
New version: there are no impl classes anymore in the new version

#18
Problem: ApplicationFactory.registerAtRegistry:
 method does not belong here
PL: MP -> HC, KISS => MIMC
New version: solved
Recurring: registerAtMessageQueue

#19
Problem: ApplicationFactory.main registers store and
reporting/enterprise component
PL: HC
New version: solved

#20
```

```
Problem: ApplicationFactory makes the components globally visible
PL: LC -> RoE -> GP
New version: solved

#21
Problem: OptimisationSolverIf.solveOptimization works with Hashtables
instead of Maps
PL: DIP
New version: solved

#22
Problem: OptimisationSolverIf sould be named TransportationCostOptimizerIf
PL: PLS -> MP
New version: still present

#23
Problem: AmplStarter should be named AmplTransportationCostOptimizerImpl
PL: PLS -> MP
New version: still present but beter/different

#24
Problem: cplex_parser in separate package in separate jar
but tightly coupled with problem to solve
PL: LC
New version: solved

#25
Problem: The optimizer should work out the optimization criteria itself
PL: TdA/IE -> IH/E, HC, MP | IH/E -> EUHM, KISS | HC -> ECV | EUHM -> IAP
New version: still present

#26
Problem: The optimizer is an internally used class and should thus
use Stores and not StoreTOs
PL: MP
New version: still present

#27
Problem: getOfferedStockItemsPerStore gets Collection<Store> stores and
long[] productIds; in OO objects should be used not IDs
PL: MP
New version: still present
```

#28
Problem: getStoreDistances(Collection<Store> stores, StoreTO callingStore) uses Store and StoreTO
PL: UP -> PLS
New version: solved

#29
Problem: storequery.queryStoreById(storeTO.getId(), pctx); There should be a method for making a TO to a DO and the clases should take a DO as parameter not an ID
PL: TdA/IE
New version: still present
Recurring: several similar constructs all over the code

#30
Problem: ReportingImpl.append(): name does not tell what it does
PL: PLS -> MP
New version: solved

#31
Problem: doubles are used for representing prices; better use a Money class (value object)
PL: MP -> ECV
New version: still present
Note: a specific problem with using doubles for money are rounding errors; this is dangerous in a commercial application. So better data types are int, long and best: BigDecimal. Together with a currency value this makes up a Money class. The principle languge does not treat that problem directly. It is too specific and actually rather a misuse of the programming languge than a normal design fault

#32
Problem: FillTransferObjects is a verb but a class
PL: MP
New version: still present

#33
Problem: why does StoreIf.changePrice() return a ProductWithStockItemTO? Not necessary.
PL: PLS -> EUHM -> ML
New version: still present

#34

```
Problem: StoreIf.getStore should be named asStoreTO
PL: PLS -> MP
New version: still present

#35
Problem: StoreIf.getOrder(long orderId): never used
PL: KISS -> MP
New version: still present

#36
Problem: ComplexOrderEntryTO[] getStockItems(ProductTO[]
requiredProductTOs): why does this return an OrderEntry?
PL: PLS -> MP
New version: still present

#37
Problem: markProductsUnavailableInStock() does more than it says
PL: PLS -> MP
New version: still present

#38
Problem: getProductsWithLowStock() ==> criterion is part of the
documentstion and inconsistent with implementation
PL: GP -> ECV | IH/E
New version: still present

#39
Problem: checkForLowRunningGoods() not only checks but also oders
PL: PLS -> EUHM
New version: still present

#40
Problem: calculateProductAmounts() has precondition:
currentStock <= mimStock
PL: IAP -> KISS
New version: still present

#41
Problem: Connector does some work in constructor
PL: PLS -> EUHM -> ML
New version: solved

#42
```

```
Problem: Connector is a verb class
PL: MP
New version: solved

#43
Problem: OrderButton knows TableModel
PL: MP -> KISS | LC
New version: solved

#44
Problem: OrderButton has a method ActionPerformed but is not a
Listener, so the method is not invoked automatically;
same with refreshButton
PL: PLS -> EUHM
New version: solved

#45
Problem: RefreshButton knows JTabbedPane
PL: LC | MP -> KISS
New version: solved

#46
Problem: RefreshButton.addElem should be called addrefreshable
PL: PLS -> MP
New version: solved

#47
Problem: Coordinator does nothing; Coordinator and
CoordinatorEventHanlerImpl should be merged
PL: MIMC -> HC
New version: now different

#48
Problem: PrinterController.append(): function unclear as of identifier;
PL: PLS -> EUHM -> ML
New version: solved

#49
Problem: parsing numbers is not the task of a printer
PL: PLS -> MP
New version: solved

#50
```

```
Problem: tasks of PrinterController are fuzzy; there should be
two classes one should just print and the other should decide
on what to print
PL: MP -> HC -> MIMC, LC
New version: solved

#51
Problem: CashDeskGUI.setBarcodeNotValid() unclear;
should be called showErrorInvalidBarcode
PL: PLS
New version: solved
```