Code Structure Definition and Verification

Stefan Walter[1]

Informatik III, University of Bonn, Germany
stefan.walter@uni-bonn.de

Abstract

This paper compares two different languages resp. toolkits to define and verify code structures, i.e., object-oriented design patterns. Informal descriptions of design patterns are transformed into formal notations. It is revealed that UML class diagrams are not best suited for automated verification of those code structures. Instead, the formal languages LePUS3 and Class-Z and SPQR are presented. Concrete examples are given by the factory method pattern and the adapter object pattern. The evaluation and results show that SPQR is more complex but more powerful, whereas LePUS3 and Class-Z are more easy to read and understand but still powerful enough to model most design patterns.

1 Introduction

Nowadays, source code often spans millions of lines of code and software even gets bigger and more complex. "The way" how the software was built, i.e., the structural decisions are specified in most cases informal only. A tool to detect and verify such code structures serves two objects. On the one hand, it can be used to follow specifications, constraints and rules during initial development. On the other hand, it leads to a better understanding of what was done with what intention. Since software often is built from existing one this is very helpful for future developers. Thus, code structure definition and verification is useful for both: forward engineering and reverse engineering.

1.1 Code Structure and Design Patterns

When it comes to object-oriented programming design patterns become important. Those are proven code structures to solve specific problems. The well-known design patterns every programmer should know and use are defined by the Gang of Four. Gamma et al. created a collection of useful object-oriented design patterns [8]. Although those patterns were written down in 1994 they are still relevant today and an important part in the process of software development. However, the definitions of those are given in an informal way not yet possible to automated verify in some given source code. The informal descriptions serve good for communicating design patterns between developers. To have a tool support (automated verification) available a formal definition of them is needed, though. Therefore, this paper discusses and compares LePUS3 and Class-Z [5] and SPQR [14]. These are languages and a toolkit to detect and verify design patterns in source code.

1.2 Related Work

There exist several other projects dealing with how to describe design patterns in a formal way. Similar approaches for pattern detection and verification follow IntensiVE [3], CrocoPat [2] and PINOT [12].

IntensiVE (**Intensi**onal **V**iews **E**nvironment) is a software system that supports developers during the process of software development, i.e., programming. Besides checks for correct implementation of design patterns it allows to specify rules for detection and verification of any other structural regularities, naming conventions or bad code smell. The system uses SOUL, a query language similar to Prolog. The supported programming languages to query and inspect are Java, Smalltalk, C and Cobol.

CrocoPat is an analysis tool that does sub-graph search on the source code. Every relation within the source code is represented by a BDD (**B**inary **D**ecision **D**iagram). With the *Call*, *Inherit* and *Contain* relation most patterns can be described. Therefore, the patterns to detect are described in standard relational algebra as well. CrocoPat then searches for matching patterns in the source code.

The PINOT (**P**attern **IN**ference rec**O**very **T**ool) analyzes Java code using so called basic blocks extracted from the abstract syntax tree. Those basic blocks are then linked together based on the execution flow to form a control-flow graph. A basic block consists of conditions (which must be met) and statements (which are executed under the conditions). Compared to other tools PINOT does not have to work on the entire abstract syntax tree when only a portion of a method body is needed to check for the existence of an implemented design pattern.

2 LePUS3 and Class-Z

LePUS3 and Class-Z are both modeling languages to formally specify object-oriented designs, and hence, as well design patterns. LePUS3 is a visual modeling language to create so called Codecharts. In contrast, Class-Z is a symbolic language. It uses the Schema notation borrowed from Z. Both are equivalent, meaning, there is a one-to-one mapping between LePUS3 and Class-Z. The languages are designed as design description languages. Therefore, they are formal languages based on first-order predicate logic. LePUS3 Codecharts and Class-Z Schemas are automated verifiable. The Two-Tier Programming Toolkit [9] supports the verification of design patterns in source code. Besides pre-existing design patterns, it allows to model any Codechart with LePUS3. It then can be verified if the source code implements the specific design. The Two-Tier Programming Toolkit is able to evaluate Java source code only.

The root of the analysis is the source code. However, source code usually spans thousands of lines of code, spread over multiple files, and containing implementation details not necessary for the design. Thus, abstract semantics is defined to represent an appropriate abstract representation of a program. It is a simplified view of it. Every program can be mapped to an extended finite structure called design model $\mathfrak{M} \mathrel{\widehat{=}} \langle \mathbb{U}., \mathbb{R}, \mathcal{I} \rangle$. It contains the abstract semantics of that program. $\mathbb{U}$ is the universe of $\mathfrak{M}$ and contains the union of entities of dimension 0 and entities of dimension 1. $\mathbb{R}$ is a set of unary and binary relations. They describe the relations between entities. $\mathcal{I}$ is an interpretation function. It maps some constant terms to entities in $\mathbb{U}$. An entity of dimension 0 is a class $\mathbb{CLASS}$ or method signature $\mathbb{SIGNATURE}$. An entity of dimension 1 is a powerset of classes $\mathcal{P}\mathbb{CLASS}$, powerset of method signatures $\mathcal{P}\mathbb{SIGNATURE}$, or a set of classes in a class hierarchy $\mathbb{HIERARCHY}$. All entities are described by a unary relation. A single unary relation is a set of entities of dimension 0, i.e., the unary relation *Abstract* contains all abstract classes. The relation between entities is described by a binary relation, i.e., *Inherit*$=\{\langle\ \underline{\texttt{collection}},\ \underline{\texttt{object}}\ \rangle\}$, stating that collection is a subtype of object. The binary relation is also defined for the transitive clojure, indicated by a $+$. There is no dedicated symbol or binary relation for methods in LePUS3 and Class-Z. Instead, methods are denoted using the binary operator $\otimes$, e.g., $\underline{\texttt{sig}}\otimes\ \underline{\texttt{cls}}$, where $\underline{\texttt{sig}}$ is an entity of type $\mathbb{SIGNATURE}$ or $\mathcal{P}\mathbb{SIGNATURE}$ and $\underline{\texttt{cls}}$ is an entity of type $\mathbb{CLASS}$, $\overline{\mathcal{P}\mathbb{CLASS}}$, or $\mathbb{HIERARCHY}$.

LePUS3 and Class-Z are the actual specification languages of the abstract semantics. An entity is called a term and is either a constant or a variable. To differentiate between those table 1 shows the different font styles and capitalizations used. The difference between a

Type	Constant	Variable
$\mathbb{CLASS}$	cls	*cls*
$\mathcal{P}\mathbb{SIGNATURE}$	Classes	*Classes*
$\mathbb{SIGNATURE}$	sig	*sig*
$\mathcal{P}\mathbb{SIGNATURE}$	Signatures	*Signatures*
$\mathbb{HIERARCHY}$	Hrc	*Hrc*

Table 1: Table

constant term and a variable term becomes important when it comes to design verification. A constant term matches only the implementation with the same name, whereas a variable term matches any. Besides the unary and binary ground formuluae three additional predicate formulae exist: $ALL(UnaryRelation, Domain)$, stating that every element in *Domain* is in the relation *UnaryRelation*, $TOTAL(BinaryRelation, Domain, Range)$, stating that each element in *Domain* is in relation *BinaryRelation* with some element in *Range*, and $ISOMORPHIC(BinaryRelation, Domain, Range)$, stating that relation *BinaryRelation* connects each element in *Domain* with an element of the set *Range*, where no two elements from *Domain* are in a relation with the same element of *Range*. Consequently, each $ISOMORPHIC$ relation is as well a $TOTAL$ relation.

An abstract semantics representation in Class-Z uses the schema notation of Z as shown in figure 1. It consists of two parts separated by a horizontal line. The upper part contains all the terms used in the formulae. The formulae are defined in the lower part. They have to be well-formed according to the Class-Z specification. A well-formed fomula describes constraints on terms and relations between them.

$SchemaName$

$$\begin{array}{l} declaration : \mathbb{TYPE} \\ declaration : \mathbb{TYPE} \\ \ldots \\ \hline formula \\ formula \\ \ldots \end{array}$$

Figure 1: General Schema in Class-Z

In LePUS3 every part of the abstract semantics is represented by a visual token. A class is represented by a rectangle, a signature by an ellipse and a hierarchy by a triangle. Methods are constructed by an overlay of a signature with a class or hierarchy. To define a term of a higher dimension a gray background shade is used. Besides the different font styles and capitalizations, terms that represent constants have a gray background and variable terms have a white background. A unary relation is defined by an overlay of an upside-down triangle with a term. The binary relation between two terms is represented by a simple arrow between those. An isomorphic binary relation is represented by a double arrow on one side. In LePUS3 there is no distinction between any binary relation and a total binary relation. Both are satisfied under the same conditions.

Even though LePUS3 and Class-Z can be used to model many patterns it is limited. Not all Gang of Four design patterns can be modeled. It is limited due to the fact that objects are not modeled directly but only classes. Thus, it is not possible to model the singleton pattern. Additionally exact behaviour and states cannot be modeled, like "a class A holds exactly 3 instances of type B". A complete list of design patterns that can be modeled is presented in [4].

2.1 Adapter Object

In figure 2 the UML diagram of the well-known adapter object pattern is shown. It is a structural pattern that belongs to the patterns of the Gang of Four. Whenever an existing interface

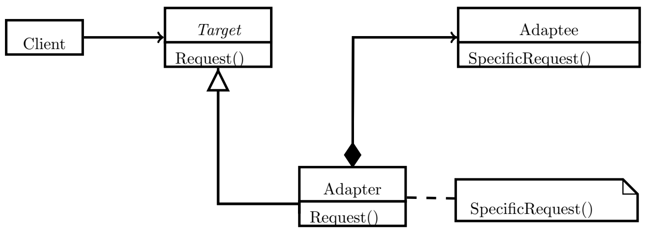

Figure 2: UML Adapter Object Pattern

has to be adapted to fit a required interface this pattern can be used. It lets work classes together that could not work together otherwise because of incompatible interfaces, i.e., a method returns boolean values as integers but real boolean values (true/false) are required. The adapter contains an instance of the adaptee and delegates the method calls to the wrapped adaptee.

The Codechart in figure 3 looks very similar to the UML diagram. Although, compared to the UML diagram there are some important differences. In the Codechart all classes and methods are variables. The UML diagram explicitly requires an abstract class called "Target". This is not the case in the Codechart. Any abstract class or even an interface fullfils the requirements. The requirements include the relations between the classes and methods they contain. Whereas the UML diagram only shows one method, in the Codechart a set of methods is shown. In this case the UML diagram could be said to be wrong, since the adapter object pattern is not restricted to deal with a single method. Additionally the Codechart contains specified relations between the classes in terms of binary relations.

Since there exists a one-to-one mapping between LePUS3 Codecharts and Class-Z Schemas the adapter object pattern written in Class-Z is straight forward. It is shown in figure 4. Every element found in the Codechart is also contained in the Schema. The classes and signatures

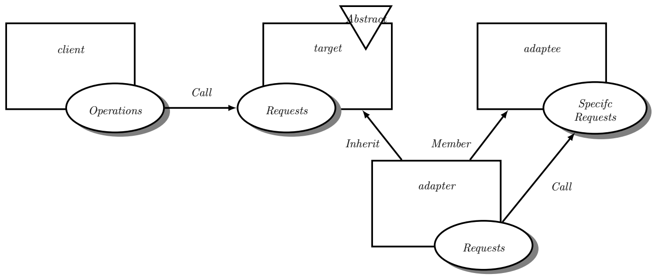

Figure 3: Codechart Adapter Object Pattern

used are defined in the upper part of the Schema. The lower part contains the well-formed formulae, i.e., unary and binary relations on the terms.

AdapterObject

$$\begin{array}{l}
client, target, adaptee, adapter : \mathbb{CLASS} \\
Operations, Requests, \\
\quad SpecificRequests : \mathcal{P}\mathbb{SIGNATURE} \\
\hline
Abstract(target) \\
Inherit(adapter, target) \\
Member(adapter, adaptee) \\
Total(Call, Operations \otimes client, Requests \otimes target) \\
Total(Call, Requests \otimes adapter, \\
\quad SpecificRequests \otimes adaptee)
\end{array}$$

Figure 4: Schema Adapter Object Pattern

2.2 Factory Method

The factory method is a creational design pattern defined by the Gang of Four. The corresponding UML diagram is shown in figure 5. It can be used if the class that creates the object shall not contain the implementation of the actual object it creates. The instantiation of the object is deferred to a subclass. Again, in the UML diagram it is only possible to state exact

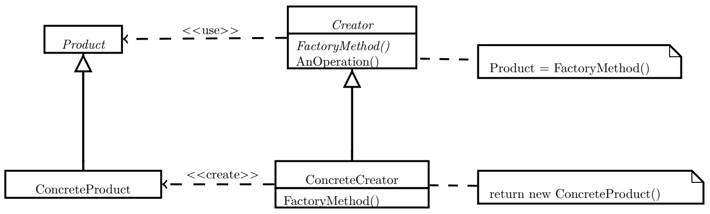

Figure 5: UML Factory Method Pattern

names of the classes. In addition, "Product" is said to be an abstract class. However, the factory method could be implemented using an interface as well.

The Codechart representing the factory method pattern in figure 6 is simpler than the UML diagram. Still, it is more precise according to the definition of the pattern. Factories and

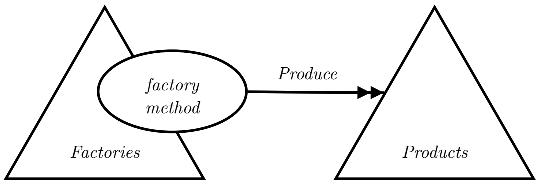

Figure 6: Codechart Factory Method Pattern

Products are hierarchy variables. They state that there is one or are many classes inheriting the exact same base class or interface. Each derived class in Factories overrides one method from the base class or interface that creates a specific product of Products. The Codechart is more general in the sense that there may be multiple factories creating different products. The isomorphic relation states that each factory must create a different product. No two products are created by the same factory.

Again, the according Class-Z Schema for the factory method pattern in figure 7 is quite easy to understand.

FactoryMethod

$Factories, Products : \mathbb{HIERARCHY}$

$factoryMethod : \mathbb{SIGNATURE}$

$Isomorphic(Produce, factoryMethod \otimes Factories,$
$\quad Products)$

Figure 7: Schema Factory Method Pattern

2.3 UML Versus Codecharts

As it was shown, Codecharts and UML class diagrams look very similar. Naturally, the question arises for the need of and the relation to Codecharts. Eden dedicates a chapter, with the same title "UML Versus Codecharts", in the book Codecharts [6] to this question. UML class diagrams are the de facto standard of the industry for software modeling. They are intuitive understandable. Compared to LePUS3, that consists of 15 tokens only, the vocabulary of UML is much more bigger. In addition, UML class diagrams can be extended in any way by stereotypes even without specifying the precise meaning. Furthermore, UML class diagrams represent implementation details, e.g., the exact names of classes and methods. Codecharts are more abstract. They do not deal with implementation details, e.g., variables are used instead of concrete class names. And the binary relation *Inherit* does not care if inheritance is implemented by using interfaces or class inheritance. All in all, UML class diagrams are neither based on a formal language nor based on a sound mathematical theory and hence not automated verifiable. As this is Eden's view on UML as a graphical language to model design patterns, there is also the other party who sees for sure the possibility and the usage of UML for design pattern modeling.

For example, France et al. [7] refer to model driven architecture and make use of metamodels to describe design patterns. Constraints that cannot be modeled in UML directly the **O**bject **C**onstraint **L**anguage is used. A design pattern is described by a pattern solution. Such a pattern solution consists of a **S**tructural **P**attern **S**pecification and an **I**nteraction **P**attern **S**pecification. Both are metamodels describing a specific design pattern by specializing the original UML metamodel for class and sequence diagrams. From an implementation (source code) a class diagram and a sequence diagram can be created. To check whether the implementation implements a specific pattern those diagrams must then conform to the SPS and IPS.

3 SPQR

The **S**ystem for **P**attern **Q**uery and **R**ecognition is a toolkit to detect and verify design patterns in source code. It describes a sequence of steps to be executed using specific tools and languages to extract those patterns. It was developed by Jason McC. Smith during his PhD studies [13]. The chain of tools of the SPQR is shown in figure 8. In the current version only

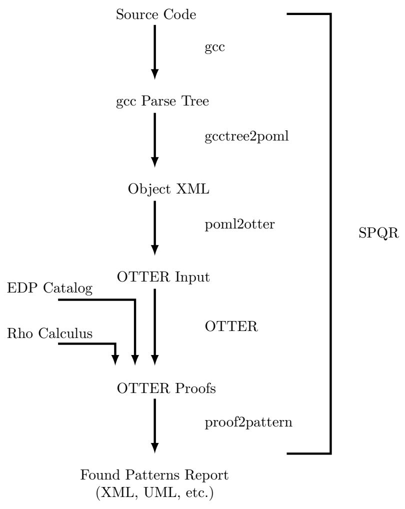

Figure 8: Structure of SPQR, cf. [14].

C++ source code can be analyzed. However, the system is designed in a language independent way. Each tool can be run stand-alone. At first, *gcc* produces a tree dump of the source code. Next, *gcctree2poml* converts the output of *gcc* into an intermediate XML format called POML (**P**ortable/**O**bject **M**arkup **L**anguage). Those two steps are language dependent. For any other programming language a converter could be written that produces a POML file from the source code. Once the XML file is generated it is converted into a file format accepted by *OTTER*. *OTTER* finds instances of design patterns with the assistance of elementary design patterns described in rho-calculus. In a final step a report is generated by *proof2pattern*.

EDPs (**E**lemental **D**esign **P**atterns) are made up of ρ-calculus. The ρ-calculus is a subset of the ς-calculus [1] extended with reliance operators. The ς-calculus is used to describe primitive objects in a formal way. It supports object formation with methods and fields and their invocation, overriding, selection, and update. Only a few concepts of the ς-calculus are needed. Specifically, these are: a type definition operator $T \equiv [...]$, where the content of the brackets defines methods and fields of the type T, a type assigning operator for objects, where $O : T$ states that object O is of type T, and an inheritance operator to declare $T\prime$ to be subtype of T, $T\prime <: T$. One more operator is introduced to form the ρ-calculus. The abstract reliance operator $<$ has three concrete types: one for method invocation ($<_\mu$), one for field access ($<_\phi$), and a generalized reliance ($<_\lambda$). Optional, there is an annotation for the method invocation reliance operator to describe a similarity association (+) or a dissimilary association (-). With this ρ-calculus concept EDPs are defined to make up the well-known design patterns.

Each design pattern is described by combining other smaller patterns or base patterns, i.e., the EDPs. A few EDPs expressed in ρ-calculus are explained in more detail. Those are needed to express the adapter object pattern and the factory method pattern. The complete list of EDPs can be found in [13].

One of the easier EDPs to understand is the **AbstractInterface** defined in equation 1. It is used to just declare a method in a class or interface without giving it a definition. How this is actually done strongly depends on the programming language, e.g., in C++ a method is assigned 0: `virtual void myMethod() = 0;`. The **AbstractInterface** takes two arguments: a class or interface and the method which will be declared to be abstract.

$$\frac{\begin{array}{c} E \vdash C, C : [l_i : B_i^{i \in 1..n}] \\ C \equiv [l_i = b_i^{i \in 1..m-1, m+1..n}, l_m = []] \end{array}}{\textbf{AbstractInterface}(C, l_m)} \tag{1}$$

In the environment E there exists an object C that has n methods and fields l_i of type B_i. Every method and field l_i has a definition b_i except for method l_m. That one is defined empty, i.e., $l_m = []$ says, it has no definition.

The **AbstractInterface** itself is not of much use. The abstract method is assigned a definition by **FulfillMethod** as shown in equation 2.

$$\frac{\begin{array}{c} \textbf{AbstractInterface}(Abstractor, operation_m) \\ ConcreteClass <: Abstractor \\ ConcreteClass \equiv [operation_i \Leftarrow b_i^{i \in 1..m-1, m+1..n}, \\ operation_m \Leftarrow b_m] \end{array}}{\begin{array}{c} \textbf{FulfillMethod}(Abstractor, ConcreteClass, \\ operation_m) \end{array}} \tag{2}$$

It fulfills an abstract method. It takes three arguments whereat *Abstractor* and $operation_m$ are as well used in **AbstractInterface**. *ConcreteClass* is defined to be derived from *Abstractor*. Each $operation_i$ gets a definition b_i with a special treat for $operation_m$ to be fulfilled by b_m. The update operator $\Leftarrow$ is used to reassign the method definitions.

A very general form of calling a method from one object in another object is described by the **Delegate** EDP in equation 3. One method delegates some work to be done to another method in another object.

$$\frac{\begin{array}{c} Delegator : [target : Delegatee, operation : B_i] \\ Delegator \equiv [operation \Leftarrow ..., target.operation2, ...] \\ Delegatee : [operation2 : B_i] \\ del : Delegator \\ del.operation <_{\mu}^{-,-} target.operation2 \end{array}}{\mathbf{Delegate}(del, target, operation, operation2)} \tag{3}$$

The *Delegator* contains the *operation* and the *target* of Type *Delegatee*. The *operation* does some work and calls *operation*2 from the *Delegatee*.

The **Conglomeration** EDP is used to conglomerate operations of a more complex task within a single object. This is also known as message decomposition. It is a specialization of the **Delegate** EDP where a method is called of the object itself. One bigger method is broken up into smaller ones. Those methods are then re-combined in another method.

$$\frac{\begin{array}{c} c : Conglomerator \\ c.operation <_{\mu}^{+,-} c.operation2 \end{array}}{\mathbf{Conglomeration}(c, operation, operation2)} \tag{4}$$

The class c contains *operation*2 and *operation*. The method *operation* calls *operation*2. Both methods are conglomerated in one object.

To use a non-local object in the local scope the **Retrieve** EDP in equation 5 is used. It creates a binding between the remote object and the local one.

$$\frac{\begin{array}{c} o.s \Leftarrow o\prime.s\prime \\ o\prime.s\prime \overset{v}{\leadsto} x \end{array}}{\mathbf{Retrieve}(o.s, o\prime.s\prime, x)} \tag{5}$$

It works in both cases regardless if $o.s$ and $o\prime.s\prime$ are methods or fields. Therefore, e.g., a field $o.s$ can be updated by a method invocation of $o\prime.s\prime$ that will return x.

At last, the most complex construct, the **CreateObject** EDP is shown in equation 6. It describes how to create and instantiate an object in ς-calculus.

$$\frac{\begin{array}{l} A \equiv \mathbf{Object}(X)[l_i \nu_i : B_i^{i \in 1..n+m}] \\ \overline{A} : \mathbf{Class}(A) \triangleq \mathbf{subclass\ of} \overline{A\prime} : \mathbf{Class}(A\prime) \\ \qquad \mathbf{with}(\mathbf{self} : \lfloor A \rfloor) \\ \qquad\quad l_i = b_i^{i \in n+1..n+m} \\ \qquad \mathbf{override} \\ \qquad\quad l_i = b_i^{i \in Ovr} \\ \quad \mathbf{end} \\ a = \mathbf{new} \overline{A} \end{array}}{\mathbf{CreateObject}(A, a)} \tag{6}$$

The ς-calculus does not have explicit notations for classes and inheritance. Hence, it is defined that the type $Class(X)$ exists if X is $Object(Y)$. Any object that is a $Class$ can then be instantiated with new. Since **CreateObject** may be called on a class in a hierarchy it does have its own methods, but may override some methods from its base class as well. All methods and fields $n+1$ to $n+m$ belong to the subclass. The methods and fields 1 to n depicted by Ovr are properties of the base class.

With this set of EDPs it is now possible to define the two Gang of Four patterns adapter object pattern and factory method pattern.

3.1 Adapter Object

The adapter object pattern should be very easy to understand now. It consists of two EDPs only. Comparing the UML diagram in figure 2 and equation 7, all objects and methods appear in both versions. Only the client is left out in the equation.

$$\frac{\begin{array}{l}\textbf{Delegate}(Adapter, Adaptee, Request, SpecificRequest)\\ \textbf{FulfillMethod}(Target, Adapter, Request)\end{array}}{\begin{array}{l}\textbf{AdapterObject}(Adapter, Target, Request, Adaptee,\\ \quad SpecificRequest)\end{array}} \tag{7}$$

Adapter is inherited from *Target* and overrides the method *Request*. The *Adapter* then calls *SpecificRequest* in the *Adaptee* from *Request* as described by the **Delegate** EDP.

3.2 Factory Method

The equation 8 showing the factory method pattern is a bit longer. But still, it should be straight forward to understand. All six classes and methods appearing in the UML diagram 5 do appear in the **FactoryMethod** as well. Unfortunately, there is a slight difference in the use of the **CreateObject** EDP compared to its definition. However, this is how it is defined and used in [13].

$$\frac{\begin{array}{l}\textbf{Conglomeration}(Creator, AnOperation,\\ \quad FactoryMethod)\\ \textbf{FulfillMethod}(Creator, ConcreteCreator,\\ \quad FactoryMethod)\\ Creator.product : Product\\ ConcreteCreator.FactoryMethod \rightsquigarrow retVal\\ \textbf{Retrieve}(Creator.AnOperation,\\ \quad Creator.product, retVal)\\ ConcreteProduct <: Product\\ \textbf{CreateObject}(ConcreteCreator.FactoryMethod,\\ \quad ConcreteProduct, retVal)\end{array}}{\begin{array}{l}\textbf{FactoryMethod}(Creator, ConcreteCreator,\\ \quad AnOperation, FactoryMethod, Product,\\ \quad ConcreteProduct)\end{array}} \tag{8}$$

4 Comparison

LePUS3 and Class-Z and SPQR are able to find design patterns in source code. However, there are some (important) differences: first of all the up-to-dateness and further research. The latest publication about LePUS3 and Class-Z is the book written by Eden in 2011 [6]. Although there are some newer publications [11], there have not been done any further improvements or changes since then. Unfortunately, the latest literature available about SPQR is the dissertation of Smith from 2005 [13]. This is as well the only paper which describes SPQR. LePUS3 and Class-Z were developed over some years, starting with LePUS (the first version).

SPQR is more complex but modularized. Hence, it is (more) easy to support additional programming languages for SPQR. On the other hand, an advantage of LePUS3 (Class-Z) is its simplicity. However, it lacks the possibility to add new patterns due to its simplicity, i.e., its restriction to a fixed set of 15 kinds of tokens only. Adding new patterns to SPQR is possible by adding new rules, i.e., adding new EDPs. It is easier to understand Codecharts since they are a visualization of design patterns. LePUS3 resp. Codecharts and Class-Z were developed due to the inexistent mathematical formalism of UML. In contrast, SPQR does not even try to give a visual representation of design patterns. So this is a complete different approach. Another drawback of the simplicity of LePUS3 and Class-Z is its limitation. LePUS3 and Class-Z can only model and detect a subset of patterns compared to SPQR.

The output is different as well. For LePUS3 and Class-Z there exists only the Two-Tier Programming Toolkit which loads Java source code and detects or verifies it against some patterns in-program. In comparison, SPQR can only be used to analyze C++ source code. It generates an XML report of found patterns. The Two-Tier Programming Toolkit can be used to explicitly check whether a program contains a specific pattern.

5 Future Work

There are many tools for design pattern detection available. Two of them were considered in more detail in this paper. Each of the tools has its own stengths and weaknesses. Precision and recall of detecting design patterns could be improved when combining the results from different tools. One problem are the different output formats each tool provides. Therefore, Kniesel et al. propose DPDX [10] (*not officially stated*: **D**esign **P**attern **D**etection e**X**change format). DPDX is a rich common exchange format for design pattern detection tools. It consists of three meta-models which add up to the DPDX format: the design pattern schemata meta-model, the program element identifier meta-model and the DPD results meta-model. For the ease of use and long-term maintainability the implementation of the models is based on simple XML. Unfortunately, there is no known reference implementation of DPDX available yet.

6 Conclusion

In this paper two different approaches and tools to detect and verify design patterns were compared. By means of the factory method pattern and the adapter object pattern explicit examples were given to show how both approaches define design patterns in a formal way. LePUS3 and Class-Z aim for an easy understandable representation and visualization of design patterns. SPQR is a more complex toolkit to define design patterns in a special calculus called

ρ-calculus. It was revealed that UML is not the best way to describe design patterns formally. The presented tools LePUS3 and Class-Z and SPQR do a good job in pattern detection. Both have some advantages and disadvantages compared to each other. However, they are fully functional and usable. A better result would be obtained when combining several results of different tools. Therefore, the interchangable DPDX format was proposed by some researchers.

References

[1] Martn Abadi and Luca Cardelli. *A Theory of Primitive Objects.* Springer, New York, 1996.

[2] Dirk Beyer and Claus Lewerentz. CrocoPat: A tool for efficient pattern recognition in large object-oriented programs. Technical report, Technical University Cottbus, January 2003.

[3] Johan Brichau, Kim Mens, and Andy Kellens. Enforcing structural regularities in source code using IntensiVE. In *Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 471–472. IEEE Computer Society Press, 2008.

[4] Amnon H. Eden, Epameinondas Gasparis, and Jonathan Nicholson. The 'Gang of Four' companion: Formal specification of design patterns in LePUS3 and Class-Z. Technical Report CSM-472, ISSN 1744-8050, Department of Computer Science, University of Essex, December 2007.

[5] Amnon H. Eden, Epameinondas Gasparis, and Jonathan Nicholson. LePUS3 and Class-Z reference manual. Technical Report CSM-474, ISSN 1744–8050, Department of Computer Science, University of Essex, December 2007.

[6] Amnon H. Eden and Jonathan Nicholson. *Codecharts: Roadmaps and Blueprints for Object-Oriented Programs.* Wiley-Blackwell, April 2011.

[7] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-based pattern specification technique. *IEEE Transactions on Software Engineering*, pages 193–206, 2004.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[9] Epameinondas Gasparis, Jonathan Nicholson, and Amnon H. Eden. LePUS3: An object-oriented design description language. In *Proceedings of the 5th international Conference on Diagrammatic Representation and Inference*, Diagrams '08, pages 364–367. Springer, 2008.

[10] Günter Kniesel, Alexander Binun, Péter Hegedus, Lajos Jeno Fülöp, Alexander Chatzigeorgiou, Yann-Gaël Guéhéneuc, and Nikolaos Tsantalis. DPDX - towards a common result exchange format for design pattern detection tools. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 232–235. IEEE Computer Society Press, 2010.

[11] Jon Nicholson, Amnon H. Eden, Rick Kazman, and Epameinondas Gasparis. Automated verification of design patterns: A case study. *Science of Computer Programming*, 80, Part B:211–222, February 2014. Originally published online (early access) in May 2013.

[12] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 123–134. IEEE Computer Society Press, 2006.

[13] Jason McC. Smith. *SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns from Source Code.* Dissertation, University of North Carolina at Chapel Hill, 2005.

[14] Jason McC. Smith and David Stotts. SPQR: Flexible automated design pattern extraction from source code. In *Proceedings of the 18th IEEE/ACM International Conference on Automated Software Engineering*, ASE '03, pages 215–224. IEEE Computer Society Press, 2003.