6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Understanding Design Patterns as Constructive Proofs*

Szabolcs Márien[a], Gábor Kusper[b]

[a] Department of Computer Science, University of Debrecen e-mail: mariensz@delfin.klte.hu

[b] Department of Computer Science, University of Debrecen e-mail: gksuper@delfin.klte.hu

Abstract

Design Patterns have a quickly expanding literature. A big part of the literature tries to formalize Design Patterns. These works set up a language that is suitable to describe object oriented properties and formalize several Design Patterns in this language. We follow the same pattern. First we set up a very simple set based language and formulize the Abstract Factory design pattern. To be correct, we formalize the first part, the problem specification part of Abstract Factory. The second part, the problem solution, is usually overlooked in the literature. We formalize the problem solution part, too. We understand the problem specification of a design pattern as an existentional logical form and the problem solution as a constructive proof. The problem specification states that some special class or object exists and the problem solution gives a recipe how to construct this class or object. Our set based language is suitable to formalize the constructive proof, which is hidden in the GoF book textual formulation.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Design Patterns

Key Words and Phrases: Design Pattern, Formal specification of Design Patterns

1. Introduction

What was the reason of appearance of the Design Patterns in software engineering? After the object oriented technologies were widely used, there weren't

*Sponsored by Upper Austrian Government (scholarship), FWF SFB F013 (P1302) and Austro-Hungarian Action Foundation.

guarantees for that, the founded solutions for the rising problems meet the requirements of the object oriented designing solutions. The expectation is that, expert designers will have used similar proven designs to resolve similar problems in different application domains [7]. How can be collected this designer professional knowledge in other to avoid the designer failures?
The profession recognized the necessity of that the same designing cases should have to be recognized and provide for those standardized designing solutions. The results of general designing solutions are the Design Patterns. A Design Pattern reflects a generic aspect of a particular system. The collection of them (list of 23 Design Patterns) is specified in the [5].

2. Key issues of Design Patterns

2.1. Formal specification of Design Patterns

The aims, arguments and objections of the formal specifications are very well collected in the [2]. Some of these collections can be found in the following examples
Objections: [2]

Formal specifications contribute little or nothing to the understanding when and how to use Design Patterns.
Patterns are abstractions, or generalizations, and therefore are meant to be vague, ambiguous and imprecise. If they were specified in a precise form, or expressed in mathematical terms, they would no longer be patterns [3].
Arguments: [2]

It's a primarily a matter of opinion, or personal intuition, weather formalisms are important or not. Nevertheless, ambiguous descriptions are obstacle in resolving details of implementations, which require complete and accurate understanding of the solutions.
It's wrongly assumed that exact or formal specifications can only describe concrete entities, and in order to be general one has to be vague. A specifications can be precise and general at the same time.
Which elements have to comprise the Design Pattern specifications ?

There are decided information about a Design Pattern - accordingly of the convention of specification by natural languages for example in the [5] -, which a Design Pattern specification has to comprise.
Required elements (accordingly the [5]):
Patterns Name and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, Related Patterns
What was the reason of the introduction of new formalization languages?

Bypassing the ambiguous interpretation of the natural languages. The specifications of the Design Patterns are specified with natural language details in the [5], which are not accurate specification forms.
Avoiding the narrowing side-effects.

2.2. Collaborations of Design Patterns

As mentions above, using the collaboration facilities of patterns, the all design can be constituted as patterns' construction.
There are lots of compartments of the exists languages for the specifications of the collaborations. As [2] showed, all of the most important resent specification languages have compartments for analyzing this aspect of Design Patterns. The way of these researches are enforced by the components based architectures, because on that part of the developments the discovering the collaborations of the patterns base the system components architect, therefore the risk of the development is reduced with avoiding the big blunders.

3. Problem statement

3.1. Which properties of patterns can be caught by formalization?

There are lots of formal specification languages, with which the Design Patterns can be defined. However, one of the most important aim of these languages the formal specification of the Design Pattern's properties and manners, there is very hard to define so formal language, which can be so ordinary, which doesn't restrict the generality of the Design Patterns. In some researchers' opinion: If "the basic structure is fixed... this isn't pattern any more."[4][2].

3.2. There isn't facility for proofing the logical completion of the Design Pattern

The primary aim of the creation of the specification languages was the bypassing the ambiguous, which is rising when natural language is used for it. The other main aspect of these solutions is that, the specified form of the Design Patterns consists of the all property of the pattern. There isn't any intention with these forms.

3.3. Excavation of Design Patterns can't be automated

The excavation of the Design Patterns is very hard. There isn't any solution for take it easily at present. However the promotion the controlling of the competence of the Design Patterns' introduction during the designing tasks is possible.

Reason of it: There isn't any formalization method, on which's result statements a proofing methodology can be applicable.

3.4. There isn't facility for detection of the Design Patterns in the design phase

Formal specifications contribute little or nothing to the understanding when and how to use a pattern [2].

There isn't existence method for promotion the recognizing of the Design Patterns during the designing tasks. However the Patterns Detection Language (was realized by Albin-Amiot and Guéhéneu[1]) can help formalizing patterns and developed build on PDL a source-to source transformation engine, with which the patterns are detected in the source code, and if there are some mistakes in it, then the engine could repair those. So, the name of the Patterns Detection Language is ambiguous, because this specification language isn't solve the problem of detection of the Design Patterns in the design phase. It "just" can convert the knowledge source to PDL and PDL to source.

If the automatic detection of the Design Patterns had to be solved in the design phase, then the patterns would have to be caught in the model's script. In this case the cost end effort of development would been significantly reduced. Since the designing failures can be repaired during the designing phase.

3.5. Conclusion

The resent specification languages can formalize the patterns by some additional language, graphical, formalization tool, but there isn't support for excavations of the Design Patterns. The reason of it is that, just the excavated Design Patterns can be formalized, and detected, but there isn't facility for proofing the formalized statement, with which the excavation of the Design Patterns in the designing phase can be established.

4. Proposed Solution

4.1. Determination of the general properties

Which properties of Design Patterns can be specify without violation of the generality of the patterns? This is the logical form of the patterns as existential logical form.

There is a specification language, with which the patterns can be specified as existential logical form, this is the LePUS [6].

First we set up a very simple set based language and formulize as example the Abstract Factory design pattern. To be correct, we formalize the first part, the problem specification part of Abstract Factory. We understand the problem specification of a design pattern as an existential logical form. The problem specification states that some special class or object exists

4.2. Problem solution

The second part, the problem solution, is usually overlooked in the literature. We try to formalize the problem solution, too, as a constructive proof. As mentioned above the problem specification states that some special class or object exists and the problem solution gives a recipe how to construct this class or object. Our

set based language is suitable to formalize the constructive proof, which is hidden in the [5] book textual formulation.

4.3. Design phase Design Pattern excavation

The supplement of the logical proofing of the formalized pattern (as mentioned in the previous paragraph) is necessary for realization of a design phase Design Pattern excavation system, which can discover that components of the designs, which's formalized forms are logically right. This excavation method can be independently from the specific information, because the logical statement doesn't consist of case specific information.
The collected logical formulates of a design model's design components consists of lots of logical forms, and some of them may be not excavated Design Patterns' "footprints", and therefore the excavation of these new possible Design Patterns are more easily.

4.4. Design phase Design Pattern detecting

We invented a logical form of design patterns, on which with a language mapping the patterns detection can be solved in the designing phase. Using the mapping between the UML script of the design and the formulates and using the logical formulates of the Design Patterns, the design phase Design Pattern detecting can be automated and realized.

5. Introduction the specification language

In the literature we can found languages, which are suitable to describe the problem specification part of design patterns.

We know that each design pattern consists of two parts, problem specification and problem solution. Our goal is not to formalize all design patterns, but formalize the problem solution part of Abstract Factory (AF) design pattern.

We know that a class consists of three parts, set of its methods, set of its fields and initial state. An object consists of three parts, set of its methods, set of its fields and actual state. The state of the object is the value if its fields. Methods of an object operate on its fields. A method consists of four parts, name, formal parameter list, return type, which is a class, and implementation. An interface is a set of methods. An abstract class consists of three parts, set of its methods, set of its fields and an initial state, but there are some methods, which have no implementation. Since AF is a class creational pattern we do not need to deal with objects to be able to formalize AF. So it is enough to deal with only classes, abstract classes and interfaces. Since we need no objects we do not need to deal with the initial state in classes and abstract classes. Since we do not need the initial state we do not need to deal with fields. This means that classes, abstract classes and interfaces are just set of methods.

We know that methods consist of four parts, name, formal parameter list, return type, which is a class and implementation. Since we do not need states to formalize AF we do not need the implementation of methods. Without lost of generality we assume that in a set of methods each method has unique name. Hence, we do not need the formal parameter list part of methods. This means that for our purpose it is enough to define method as the ordered pair of its name of return type, which is a class.

Definition 5.1. (Name)
N is a name if and only if N is a final set.

Definition 5.2. (Method)
M is a method if and only if M = (N, C), N is a name and C is a class.

Definition 5.3. (Class or Abstract Class or Interface)
C is a class (or abstract class or interface) if and only if C is a set of methods.

Definition 5.4. (GetName, GetReturnType)
GetName(M) = N and GetReturnType(M) = C, where M is a method, N is Name, C is a class and M = (N, C).

Definition 5.5. (GetNames)
GetNames(C) = {GetName(M) | M in C}, where C is a class.

We define the Simp language.

Definition 5.6. (Simp Language)
A sentence is a sentence in Simp langue if and only if it is a first order logical formula, which may contain the name, method and class (or abstract class or interface) predicate and the GetName, GetReturnType and GetNames functions and the usual set connectives, i.e., set of ({}) quantifier, subset ($\subseteq$), element ($\in$)and equal (=) predicates and the power set function (P).

We define the basic notion, subclass, of object-oriented programming.

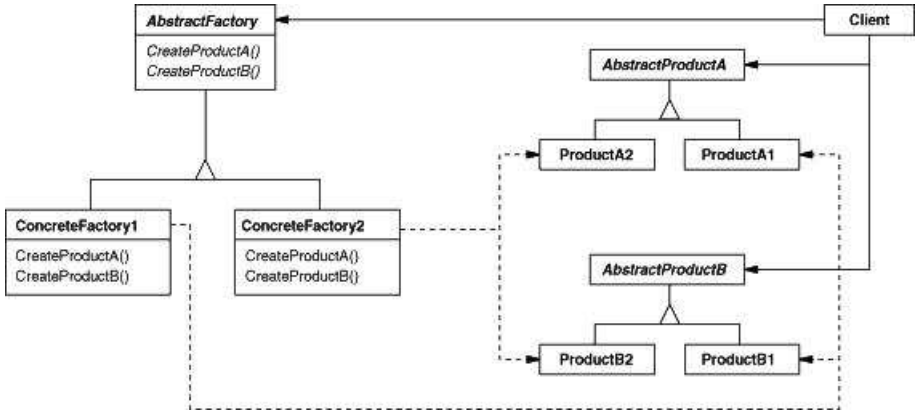
Definition 5.7. (Subclass or Implement)
D is a subclass of C (or D implements C) if and only if C is a subset of D or
D is a subclass of C (or D implements C) if and only if GetNames(C) is a subset of GetNames(D) and for all K in C there is M in D such that GetName(M) = GetName(K) and GetReturnType(M) is a subclass of GetReturnType(K).

Definition of subclass is a recursive definition. There are two variant of subclass. The first variant describes the simple case, when the ancestor class is the subset of the child class. This means that we inherit the child class from the ancestor by adding new method to it. The second variant of subclass describes the more complicated case, when we inherit the child class from the ancestor by overriding some methods (of course we can add also new methods, too). This case we allow only that the return type of the overridden method is a subclass of the return type of the original method.

Note that the definition of subclass is a sentence in Simp language. Hence, we can use the subclass predicate in Simp language sentences.

6. The Abstract Factory Design Pattern

The Abstract Factory Design Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. [5]



As each design pattern AF consists of two parts, problem specification and problem solution. Since AF is a class creational pattern we can formulize its problem specification part using the Simp language. We give this Simp sentence as a lemma and we try to prove it. We will see that we need the idea described in the problem solution part to be able to construct the proof.

Lemma 6.1. (Abstract Factory)
For each set of class APS there is an interface AF such that for each class CF which implements AF we have that for each class AP in APS there is a method M in CF such that its return type is a subclass of AP.

The same lemma using formal notation:

Lemma 6.2. (Abstract Factory)
$\forall$APS, APS is a set of classes $\exists$AF, AF is an interface $\forall$CF, CF implements AF $\forall$AP, AP $\in$ APS $\exists$M, M $\in$ CF : GetReturnType(M) is a subclass of AP.

If we try to prove this lemma we face the following problem. Using the usual automated theorem proving techniques we do the following steps. First we eliminate the first connective in the goal, i.e., the $\forall$APS, APS is a set of classes part, by fixing APS to an arbitrary set of class. Now we have in or knowledge base that APS is a set of classes and or goal is the remaining formula, i.e., $\exists$AF, AF is an interface $\forall$CF, CF implements AF $\forall$AP, AP $\in$ APS $\exists$M, M $\in$ CF : GetReturnType(M) is a subclass of AP. Now we have no other choice, we have to eliminate the first connective in the goal, which is an existentional quantifier. This is usually a difficult step. To eliminate exists from goal we need either human creativity or a powerful automated theorem proving software, which systematically creates more

and more complicated constant and investigates whether it is a solution or not. This process may take an enormous time. But in our case we do not need this process, because the second part of the design pattern, the problem solution, tells us how to construct this interface.

The problem solution part of Abstract Factory design pattern suggests that the AF (Abstract Factory) interface should contain for each Abstract Product (AP in APS) a method, which creates that product, i.e., AF should have the property: $\forall$AP, AP $\in$ APS $\exists$M, M $\in$ AF : GetReturnType(M) = AP. We use this suggestion in the proof.

Proof of Abstract Factory
To show that $\forall$APS, APS is a set of classes $\exists$AF, AF is an interface $\forall$CF, CF implements AF $\forall$AP, AP $\in$ APS $\exists$M, M $\in$ CF : GetReturnType(M) is a subclass of AP, we assume that APS_1 is a set of classes and show that $\exists$AF, AF is an interface $\forall$CF, CF implements AF $\forall$AP, AP $\in$ APS_1 $\exists$M, M $\in$ CF : GetReturnType(M) is a subclass of AP. To show this we have to find a suitable choice for AF. We use the suggestion of the problem solution part of Abstract Factory. Let AF_1 be an interface such that $\forall$AR, AR $\in$ APS_1 $\exists$N, N $\in$ AF : GetReturnType(N) = AR. We show that AF_1 is a suitable choice for AF. To show this we assume that CF_1 implements AF_1 and show that $\forall$AP, AP $\in$ APS_1 $\exists$M, M $\in$ CF_1 : GetReturnType(M) is a subclass of AP. To show this we assume that AP_1 $\in$ APS_1 and show that $\exists$M, M $\in$ CF_1 : GetReturnType(M) is a subclass of AP_1. To show this we have to find a suitable choice for M. We know that $\forall$AR, AR $\in$ APS_1 $\exists$N, N $\in$ CF_1 : GetReturnType(N) = AR. By substituting AP_1 in AR we know that for some N_1 $\in$ CF_1 we have GetReturnType(N_1) = AP_1. We know that CF_1 implements AF_1. From this, by definition of implement, we know that $\forall$K, K $\in$ AF_1 $\exists$L, L $\in$ CF_1 : GetName(L) = GetName(K) $\wedge$ GetReturnType(L) is a subclass of GetReturnType(K). By substituting N_1 in K we obtain that for some L_1 $\in$ CF_1 we have GetReturnType(L_1) is a subclass of GetReturnType(N_1). >From this and from GetReturnType(N_1) = AP_1, we know that GetReturnType(L_1) is a subclass of AP. Hence, L_1 is a suitable choice for M. Hence, $\forall$APS, APS is a set of classes $\exists$AF, AF is an interface $\forall$CF, CF implements AF $\forall$AP, AP $\in$ APS $\exists$M, M $\in$ CF : GetReturnType(M) is a subclass of AP.

Our goal was to show that using the suggestion of the problem solution part of AF we can produce the proof of the formula, which describes the problem specification part of AF. The proof uses the standard automated theorem proving techniques, no big ideas or shortcuts. Therefore, it is a bit hard to read.

References

[1] Albin-Amiot, H.; Guéhéneuc, Y.: *Meta-Modeling Design Patterns: Application to Pattern Detection and Code Analysis*, Workshop on Adaptive Object-Models and

Metamodeling Technoques. ECOOP(European Conference on Oriented Programming), 2001.

[2] Aline Lúcia Baroni: *Design Patterns Formalization. Resource Riport 03/03/Info*, Computer Science Department of École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes, 2003.

[3] Buschmann, R.; Meunier, R.; Rohnert, H,; Sommerlad, P., Stal, M.: *Pattern-Oriented Software Architecture - A System of Patterns*, Wiey and Sons, 1996.

[4] Coplien, J.O.: *Code Patterns. The Smalltalk Report*, SIGS Publications, 1996.

[5] Gamma,E; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, 1995.

[6] H. Eden, Amnon; Gil, Joseph (Yossi); Hirshfeld, Yoram; Yehudai, Amiram: *Towards a Mathematical Foundation For Design Patterns.*

[7] Dong, Jing: *Design Component Constracts: Modeling and Analysis of Pattern-Based Composition*, Ph.D Thesis, Computer Science Department of Waterloo, 2003.

Postal addresses

Szabolcs Márien
Department of Computer Science,
University of Debrecen,
2, József Attila Street,
Nyékládháza, H3433
Hungary

Gábor Kusper
Department of Computer Science,
University of Debrecen,
15, Bethlen GáborStreet,
Eger,
Hungary