Pre- and Postconditions for Refactoring to Design Patterns

Péter Csontos

Department of Computer Science, Eötvös Loránd University
Pázmány Péter sétány 1/D, H-1117 Budapest, Hungary
csonti@inf.elte.hu

Abstract. The starting point of generative and component-based programming is capturing domain knowledge and practices. Among several other techniques, refactoring to design patterns can be useful means for gathering industry best practices and describe domain knowledge. Patterns are usually combined when specifying the ideal solution for a problem. To ensure quality of multi-pattern solutions, it is viable to define pre- and postconditions for patterns. As a consequence of this, the relevance of pattern-combinations can be evaluated through matching the appropriate conditions. Latter can be used for increasing performance through skipping unnecessary precondition checks, however ensuring required quality.

Keywords: refactoring, design patterns, pre- and postconditions
Classification: first year's PhD work

1 Introduction

Gathering domain knowledge in the form of data structures, algorithms and other elements is an essential part of generative programming [6].

After having the appropriate domain model, there are several techniques available for creating highly configurable active libraries, such as aspect-oriented programming [11] (for example AspectJ [12]) and C++ template metaprogramming [5].

It would be highly useful when capturing domain knowledge from legacy systems to have the side effect of possessing the parts of these legacy systems aimed for reuse in the form of a well defined, easily maintainable and configurable software artifact. Such an artifact could be a combination of design patterns [8] [9] or an implemented form of this combination.

In order to avoid a confusing multitude of these pattern combinations, strict rules should be introduced that tell which combinations are relevant and which are not or not enough. It can be reached through defining pre- and postconditions for design patterns and matching these conditions when combining patterns.

2 Refactoring to Patterns

For extracting these combinations of patterns from existing code, let us use refactoring to patterns [10].

Refactoring [7] is a widely used technique mainly used as a part of extreme programming[3] and agile programming [4]. According to its definition by Martin Fowler, it is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [7]. In [10] Joshua Kerievsky proposes well-defined ways of transforming commonly used but not ideal program-parts to design patterns.

The result of transforming a system or a part of a system is a combination of patterns. To ensure good quality of this combination it is useful to explore the possibility of using pre- and postconditions for design patterns.

3 Pre- and Postconditions for Design Patterns

Each design pattern should have the following elements as parts of its description:

- name and intent;
- problem and **context**;
- force(s) addressed;
- abstract description of structure and collaborations in solution;
- positive and negative **consequence(s)** of use;
- implementation guidelines and sample code;
- known uses and related patterns.

In contrary with the above, context and consequences of use (to put it the other way around, pre- and postconditions) are rarely detailed enough, or if they are, they **must** be too general because design patterns are by definition language independent.

However, when refactoring patterns from existing code, the design pattern combinations are produced in the form of program code of some language. It enables us to define precise pre- and postconditions attached to each design pattern.

For example, a class matching the singleton pattern mustn't have any public constructor (in languages such as C++ or Java).

These conditions are heavily dependent on the structure (for example the type system) of the language used. It is obviuos that we get different kinds of pre- and postconditions using Smalltalk or Ada. Lots of legacy systems exist that were created using "legacy" languages such as C, COBOL, Pascal. These systems are not even object-oriented, so creating transformations between them and OO design patterns bring in further complexity.

Thinking further, if we have two collaborating patterns, we can pair a postcondition with the matching precondition of the connected pattern. Getting different results of this pairing for different pattern combinations, we can exactly

and precisely state which design patterns can work well together and which cannot.

For example (from [13]), we can combine singleton and double-checked locking optimization patterns in order to provide efficient but synchronized access to the singleton object concurrently for simultaneous threads. In this case we can state that the result (postcondition) of applying the singleton pattern (guarantee that there can only be one instance of the singleton class) is matching the requirement of the double-checked locking optimization pattern that has a precondition of having just one object of the type that has to be shared among concurrent threads. This combination is sometimes called threadsafe singleton pattern.

Another example (also from [13]) could be a strategy selection factory method. Here we've got a strategy pattern resulting in one or more strategy classes with existing public constructors (postcondition) and a factory method pattern which requires classes instantiated by the method having public constructors (precondition). Because the two conditions match, it is a natural thought to combine these two design patterns this way.

4 Further Work

Utilizing these natural matches significant performance improvement can be achieved through skipping precondition checks for pattern pairs that are formally proven to collaborate seamlessly.

Checking if a pattern can be used in a given situation, runtime techniques such as reflection can be used. Reflection is available as part of the language in Java. Concerning C++ [14] reflection is not a default language feature, but there are techniques such as [2] that make accessing metainformation on classes possible.

In C++, checking variuos properties of classes is also possible during compile-time. This static interface checking is made possible through using template metaprogramming [1].

References

1. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
2. Attardi, G., Cisternino, A.: Self Reflection for Adaptive Programming. D. Batory, C. Consel and W. Taha (Eds.): GPCE 2002, LNCS 2487, Springer Verlag (2002)
3. Beck, K., Fowler, M.: Planning Extreme Programming. Addison-Wesley (2000)
4. Cockburn, A.: Agile Software Development. Addison-Wesley (2001)
5. Coplien, J.: Multi Paradigm Design for C++. Addison-Wesley (1998)
6. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)
7. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)

8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Abstraction and Reuse of Object-Oriented Design. O. M. Nierstrasz (Ed.): ECOOP 1993, LNCS 707, Springer Verlag (1993)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
10. Kerievsky, J.: DRAFT of Refactoring To Patterns. Industrial Logic (2003). http://www.industriallogic.com/papers/rtp017.pdf
11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. M. Aksit, S. Matsuoka (Eds.): ECOOP 1997, LNCS 1241, Springer Verlag (1997)
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G.: An Overview of AspectJ. J. L. Knudsen (Ed.): ECOOP 2001, LNCS 2072, Springer Verlag (2001)
13. Schmidt, D.: Object-Oriented Design Case Studies with Patterns and C++. Vanderbilt University (2003) 87–98, 138–149. http://www.cs.wustl.edu/~schmidt/
14. Stroustrup, B.: The C++ Programming Language. Third Edition and Special Edition. Addison-Wesley (2000)