ISSN No. 0976-5697

Volume 2, No. 3, May-June 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Towards Pattern-Based Refactoring: Abstract Factory

Davoud Keshvari Ghourbanpour*
Department of Computer Science and Engineering,
Islamic Azad University of Arak
Arak, Iran
davidkeshvari@yahoo.com

Mohammad Hossein Yektaie
Faculty Member of Islamic Azad University of Abadan
Abadan, Iran
mh.yektaie@gmail.com

Abstract: Many agile software development methodologies use from refactoring process for improving codes. Amount of effort and using from different types refactoring techniques for improving the structural models of the system vary in different projects, depending on time and knowledge of developers and it's not specified how much must be done for refactoring in order to deodorizing existing bad smells in codes, clearly. Using design patterns can help refactoring process as target of refactoring for avoiding unclear process. So, when and how to apply the design patterns in codes by pattern-directed refactoring is very important. We intent to show when and how we can refactor codes towards using Abstract Factory design pattern and present a mechanics for that in agile software development methodologies where design models are lacking in detail. The presented mechanics cause low cost and clear refactoring process.

Keywords: Agile software development methodology; design patterns; code refactoring

I. INTRODUCTION

Code refactoring is "a change in the internal structure of software will be easier to understand and apply for cheaper change without a change in its behavior to be observed" [5]. Many software development methodologies use refactoring in order to increase simplicity, maintainability, reusability and readability of the codes.

Agile software development methodologies (eXtreme Programming Feature Driven Development, Scrum, etc.) which they are lightweight methodologies use from refactoring more than the other methodologies [11, 7, 4]. In some agile methodologies such as eXtreme Programming (XP) refactoring process is one of the main and fundamental processes in software development, but in others such as Feature Driven Development (FDD) refactoring process is an optional process [1, 2].

Code refactoring has advantages and limitations. Code refactoring leads to change the structures of the codes in order to increase the simplicity, readability, maintainability and usability, so developers gain a better understanding of the system and confidence to develop the system, because in agile development methods, there is little documentation and the code itself contains all the details especially in XP [10, 3]. Despite the advantages code refactoring has, some software developers and software methodologies prefer not to use refactoring or less use. There are some reasons for that as follows.

• Programmers don't know when that refactoring is needed or not.
• Programmers do not know what mechanisms should be used to solve the detected bad smells.
• Time duration of refactoring is not clear and depends on the programmer's experience and the programming language that features how to do code refactoring.
• Refactoring process will be expensive process if not done correctly and systematically.

The above restrictions cause the programmers less use from code refactoring and they prefer to develop new user stories (use cases) instead of refactoring. In this paper we will show that design patterns can be used as the target of refactoring processes and show how they can be use for targeting and structuring refactoring process. We show this for the Abstract Factory pattern.

The rest of the paper is structured as follows. In section 2, we show a brief history and related works. In Section 3, we will describe refactoring process by using Abstract Factory pattern and show how to identify what areas of code need refactoring towards the Abstract Factory pattern. In Section 4, the process described in Section 3 is presented with an example to present applicability of this process. The last section shows the conclusions and future works.

II. BACKGROUND

Refactoring process is an important process in various software development methodologies and that is used for increasing the quality of the system's structures and detecting defects. And also many software development methodologies use from design patterns for solving different recurring problems in their systems. Design patterns can be used by software development methodologies (RUP, FDD, etc.) for solving design problems in their design models. [9] Presents how improve the design models of object-oriented systems by using design patterns. [5] Introduced different kinds of bad smells in codes and models and then presented some refactoring techniques to deal with them. [8] Presented a way to refactoring codes towards design patterns and showed that design patterns can be as a target for code refactoring process. He showed his idea for some design patterns. Although, most of the researches are related to identifying design patterns in codes and those help to programmers to increase their understanding from the system and codes. In this paper, we follow [8]'s approach for refactoring codes towards design patterns, and we present this idea for the Abstract Factory pattern that this can be added to [8]'s collection as technique for refactoring.

III. REFACTORING TOWARDS DESIGN PATTERNS

A. *Abstract Factory Pattern*

We introduce the Abstract Factory pattern in this section.

1) Intent

According to [6], the intent of the Abstract Factory pattern is to "provide an interface for creating families of related or dependent objects without specifying their concrete classes."

2) Applicability

According to [6], Abstract Design pattern can be used at following situations:

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations**.**

Figure 1 shows structure of the Abstract Factory pattern.

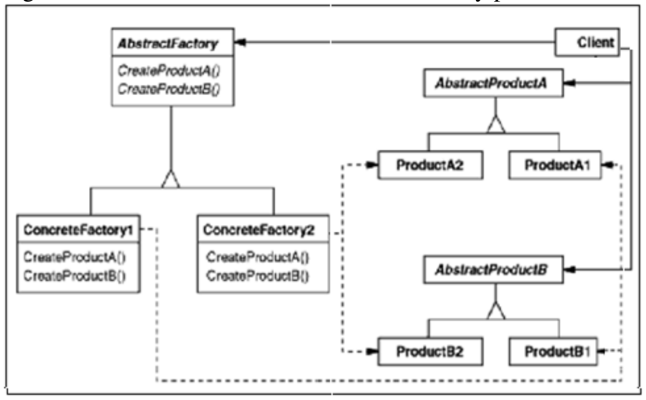

Figure 1: structure of Abstract Factory pattern [6].

3) Participants

• Abstract Factory: provides an interface for operations that produce a set of various products.
• Concrete Factory: implements the operations that present in Abstract Factory.
• Abstract Product: an interface for a type of product object.
• Concrete Product: defines a product object to be created by the corresponding concrete factory and implements the Abstract Product interface.
• Client: uses only interfaces declared by Abstract Factory and Abstract Product classes.

4) Consequences

Using Abstract Factory pattern will have the following benefits:
•it isolates concrete classes.
•it makes exchanging product families easy.
•it promotes consistency among products.
In this section, we describe code refactoring process by using Abstract Factory pattern.

When there are conditional logic statements in a method and multi related objects are created at each conditional's block, at this time it's better to refactor the codes due the following issue.

5) Problem

Client is responsible for creating a group of related objects that is caused complexity and high coupling.

If there are Switch/case or if-else clauses statements in the code and a set of related objects are created in each case's block (if's black), it's better to refactor the codes by using Abstract Factory pattern that leads to increase readability, cohesion and lowing coupling. Figure 2 shows this issue.

```
Class CA{
    Public method F1(…){
            Switch (conditionVariable) {
              Case 'X':
                    New productA1(…)
                    New productB1(…)
                    …
                     New productZ1(…)
                     Break;
              Case 'Y':
                    New productA2(…)
                    New productB2(…)
                   …
                    New productZ2(…)
                    Break;
             ….} } }
```

Figure 2: Switch/case statements and creating related objects.

B. *Mechanics*

1. First, a class for each product that is created in case's block is defined by applying Extract Class [5]. We repeat this for all objects which are created in case's block.
2. An abstract class is defined by applying Extract Supperclass [5] for the objects which have the same structure and behavior.
3. A concrete class is defined by applying Extract class [5] for each case of switch statement as concrete factory and then a method is defined in that class by applying Extract Method [5] and then the statements of the case's block are moved into that method by applying Move Field [5]. We repeat this for all cases in switch statement.
4. An abstract class is defined for all concrete classes which have been defined at step 3 as abstract factory class by applying Extract supperlcass [5] and then the required fields and operations for creating products are moved into this class by applying Move Filed [5]. The concrete factory classes inherent from this class.
5. The statements in case's block of switch statement are replaced with calling creator method of the related concrete factory class. This is done in the method of client class.

Figure 3 shows the structure of the code in Figure 2 after applying refactoring to Abstract Factory pattern.

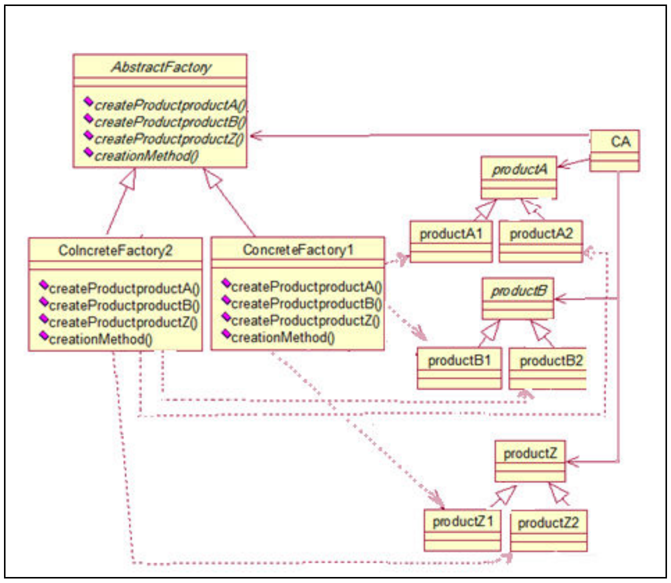

Figure 3: Structure of the code in Figure 2 after applying refactoring towards Abstract Factory Pattern

IV. EXAMPLE

An example is provided in this section to demonstrate the applicability of towards Abstract Factory patterns process for making the codes simplifier, maintainable, and reusable. We refactor the code presented in Figure 4 by using the Abstract Factory pattern.

```
class Form1
  {...
    private string DbProvider;
    private string ConnSt;
    const string DbProviderSQL="System.Data.SqlClient";
    const string DbProviderOL="System.Data.OldClient";
    Form1(){
      DbProvider = null;
      ConnSt =null
    }
   public void setConnection(string cs){ ConnSt = cs;    }
   public void setDataProvider(string dp){ DbProvider = dp; }
    void Insertrecord(){
      string commSt = "insert into table1 values('a',...,'z')";
      switch (DbProvider) {
        case DbProviderSQL:
            SqlConnection sqlconn =new SqlConnection(ConnSt);
            SqlCommand sqlcomm = new SqlCommand(commst,
sqlconn);
            sqlcomm.CommandType = CommandType.Text;
            sqlconn.Open();
            sqlcomm.ExecuteNonQuery();
            sqlconn.Close();
            break;
          case DbProviderOL:
            OleDbconnection oleconn =new OleDbConnection
(ConnSt);
            OleDbCommand olecomm = new
OleDbCommand(commSt,oleconn);
            olecomm.CommandType = CommandType.Text;
            oleconn.Open();
            olecomm.ExecuteNonQuery();
            oleconn.Close();
            break;
        }}...}
```

Figure 4: Sample Code

As we can see in Figure 4 some objects are created in each case's block. So we follow the step 1 of the mechanics in section 3, then the following classes are defined:

```
Class SqlConnection{… }
Class SqlCommand{… }
Class OleDbConnection{… }
Class OleDbCommand{… }
```

According to Step 2, interfaces are defined for related and the same type of objects as supperclass, and then the concrete product classes are inherited from the supperclass as follow:

```
Abstract Class IConnection {…}
Abstract Class ICommand {…}
Class SqlConnection: IConnection {… }
Class SqlCommand: ICommand {… }
Class OleDbConnection: IConnection {… }
Class OleDbCommand: ICommand {… }
```

Then a concrete class is defined for each case's block in codes as concrete factory class according to step 3. After that some methods are defined to create product objects and the required data are moved into these classes as follow:

```
Class SqlFactroy: AbsFactDBP {
ICommand createCommand(){ return new SqlCommand(…)}
IConnection createConnection(){ return new SqlConnection(…)}
Void execQuery(){…
sqlconn.Open();
sqlcomm.ExecuteNonQuery();
sqlconn.Close();… }
Class OledbFactroy: AbsFactDBP {
ICommand createCommand(){return new OleDbCommand(…)}
IConnection createConnection(){ return new
OleDbConnection(…)}
Void execQuery(){…
sqlconn.Open();
sqlcomm.ExecuteNonQuery();
sqlconn.Close();… }
… }
```

According to step 4 an abstract class (AbsFactDBP) is defined as abstract factory class and then the concrete factories (SqlFactroy and OledbFactroy) are inherited from the AbsFactDBP and the created methods in SqlFactroy and OledbFactroy are defined as abstract methods in AbsFactDBP too. Following codes show this.

```
Class AbsFactDBP {
        virtual ICommand createCommand(){…}
        virtual IConnection createConnection(){…}
                virtual Void execQuery();
                        …}
        Class SqlFactroy: AbsFactDBP {…}
        Class OledbFactroy: AbsFactDBP {… }
```

The statements in case's block are replaced with instantiation of the factory classes as follow:

```
class Form1  { ...
AbsFactDBP  AbsFactDProv;
    void Insertrecord(){ ...
      switch (DbProvider) {
        case DbProviderSQL:
            AbsFactDProv= new SqlFactroy();
            break;
         case DbProviderOL:
              AbsFactDProv= new OleDbFactroy();
            break;
          }}...}
```

V. CONCLUSIONS

In software development methodologies in general and agile development methodologies as particular refactoring process is use for finding defects and making codes maintainable, simplify, reusable and readable. Software developers usually use from design patterns for solving recurring design problems in object oriented designs. This article introduced refactoring process and its importance in agile software development methodologies where the design models are not rich. Code refactoring by using design patterns as target of refactoring process has been shown and a mechanics for refactoring towards the Abstract Factory pattern presented. Besides, this paper showed where and how codes can be refactored towards the Abstract Factory pattern and showed this with an example.

The proposed approach for refactoring towards Abstract Factory design patterns can be added into the [8]'s collection. In the future we can provide various refactoring mechanics for more design patterns and add them to the [8]'s collection and also we can use from the presented mechanics for refactoring test codes in automated tests.

VI. REFERENCES

[1] P. Abrahamsson, O.J. Salo, J. Ronkainen, and J. Warsta, "Agile Software Development Methods: Review and Analysis," Published by VTT, 2002.

[2] K. Beck, "Extreme Explained: Embrace, Programming Change," published by Addison-Wesley, 2004. (2nd Ed.)

[3] T. Bozheva, and M. Gallo, "Framework of Agile Patterns," in Proceedings of the SPI Conference, Europe, 2005.

[4] J. Ferreira, "Interaction Design and Agile Development: A Real World Respective," M.S Thesis, Victoria University, 2007. (Thesis)

[5] M. Fowler, Refactoring: Improving the Design of Existing Code. MacGrow-Hill, 2003.

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

[7] R. Jeffries, "Patterns and Extreme Programming," Portland Pattern Repository, Dec, 1999.

[8] J. Kerievsky, Refactorings to Patterns. Industrial logic, Inc, 2001.

[9] C. Larman, Agile Modeling with UML, Patterns and Test-Driven Development, 2008.

[10] F. Padberg, "Lean Production Methods in Modern System Development," Wirtsheafts informatik, vol. 3, no. 49, pp. 162-170, 2007. (Journal)

[11] S.R. Palmer, and J.M. Felsing, A Practical Guide to Feature-Driven Development. Prentice-Hall, 2002.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.