Design Patterns Past and Future

Aleksandar Bulajic
Metropolitan University, Belgrade, Serbia

LANB@45.dk; Aleksandar.Bulajic.1145@fit.edu.rs

Abstract

A very important part of the software development process is service or component internal design and implementation. Design Patterns (Gamma et al., 1995) provide list of the common patterns used in the object-oriented software design process. The primary goal of the Design Patterns is to reuse good practice in the design of new developed components or applications. Another important reason of using Design Patterns is improving common application design understanding and reducing communication overhead by reusing the same generic names for implemented solution. Patterns are designed to capture best practice in a specific domain. A pattern is supposed to present a problem and a solution that is supported by an example. It is always worth to listen to an expert advice, but keep in mind that common sense should decide about particular implementation, even in case when are used already proven Design Patterns. Critical view and frequent and well designed testing would give an answer about design validity and a quality. Design Patterns are templates and cannot be blindly copied. Each design pattern records design idea and shall be adapted to particular implementation. Using time to research and analyze existing solutions is recommendation supported by large number of experts and authorities and fits very well in the pattern basic philosophy; reuse solution that you know has been successfully implemented in the past.

Sections 2 and 3 are dedicated to the Design Patterns history and theory as well as literature survey. Section 4 contains General discussion and critical view and pointing to the very important warning that Design Patterns are not silver bullet. Section 5 is about Anti-patterns .Section 6 contains examples. The examples are based on the Abstract Factory design pattern and gradually demonstrate how this pattern is built and changed.

The title of this paper is “Design Patterns past and future”. Design patterns past is tightly coupled to patterns future. The past offers collection of available patterns. The future is using it as template and adapting it to the new context. The future is adding adapted patterns to existing collection and new patterns discovered by solution analyses.

Keywords: Design-Patterns, Anti-patterns, AbstractFactory,

Introduction

When in 1994, has been published a book “Design Patterns: Elements of Reusable Object-Oriented Software”, written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1994), better known as Gang of Four (GoF), it opened a new chapter in the software design and development. This book was not the first one that attracted significant attention and influenced software engineering.

In 1968 the book “The Art of Computer Programming”, written by Donald E. Knuth, was published. In 1975 “The Mythical Man-Month: Essays on Software Engineering”, written by Frederick P. Brooks, was published. In 1978 “The C Programming Language”, written by Brian W. Kernighan and Dennis M. Ritchie, was published. In 1978 and 1979 Tome De Marco published “Structured analyses and system specification”. In 1999 “Refactoring Improving the Design of Existing Code”, written by Martin Fowler, was published. Martin Folwer wrote also “Patterns of Enterprise Application Architecture”, a book published in 2003. All these books and subjects influenced significantly the software development. There are also many other important books published. An example of top hundred best software engineering books can be found on Internet at http://knol.google.com/k/top-100-best-software-engineering-books-ever# (Google Knoll, 2010).

But history of design patterns that we know it today started already in 1987. The most of following facts are available on the Internet (HistoryOfPatterns, 2010) and borrowed from this Internet site.

Ward Cunnigham and Kent Beck have been inspired by an architectural concept described in the Christopher Alexander books about reusable architecture design called patterns. Ward and Kent have studying and experimenting with patterns and presented results in 1987 at the **OOPSLA** (Object-Oriented Programming, Systems, Languages & Applications) conference in Orlando.

In the same time Erich Gamma was working on the PhD thesis about object-oriented design. He met Richard Helm in 1990 at the Bruce Anderson BoF (Birds of a Feather) session and shared common ideas about reusable software design.

At the Bruce Andersen workshop at OOPSLA 1991 conference they met Ralph Johnson, and John Vlissides and these four people becomes later known as Gang of Four (GoF).

In 1991 has been published a book “*Advanced C++ Programming Styles and Idioms”* written by Jim Coplieng, about C++ patterns which he called idioms.

“*In August of 1993, Kent Beck and Grady Booch sponsored a mountain retreat in Colorado where a group of us converged on foundations for software patterns. Ward Cunningham, Ralph Johnson, Ken Auer, Hal Hildebrand, Grady Booch, Kent Beck and Jim Coplien struggled with Alexander's ideas and our own experiences to forge a marriage of objects and patterns. We agreed that we were ready to build on Erich Gamma's foundation work studying object-oriented patterns, to use patterns in a generative way in the sense that Christopher Alexander uses patterns for urban planning and building architecture. We then used the term* generative *to mean* creational *to distinguish them from* Gamma patterns *that captured observations.*” (HistoryOfPatterns 2010). They created the Hillside Group that met again in 1994 to plan the first Pattern Languages of Programs (PLoP) conference.

Just before first PLoP conference, the Gang of Four had completed their work and sent to Addison-Wesley who rich to publish this book before conference starts.

Design Patterns: Elements of Reusable Object-Oriented Software

This book (Gamma et al., 1994) cites Christofer Alexander pattern definition:

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".* (p. 12)

The definition is followed by four elements which each pattern should contain:

1. The pattern name - which is used to distinguishes and references certain patterns, and also to create pattern language,
2. Design problem, or issue - which describes when pattern could be applied
3. The solution – example solution that is used as a template for solving design problem,
4. The consequences of applying pattern solution.

The first chapter in the book describes design patterns and reasons why this should be used in the Object Oriented Design (OOD) to solve specific design issues. According to authors definitions, design patterns are elegant and proved solutions that are already redesigned many times to provide reuse, flexibility and easy maintenance of existing source code.

The authors' purpose was recording of experience in object-oriented software as design patterns. (Gamma et al., 1994, p. 12).

More details about design patterns and how design patterns solve design problems are described in the rest of the first chapter and subchapters.

Object-oriented design problems and design patterns solution are described in the rest of the first chapter subchapters. Following is short overview of design problems where design patterns solution can be reused:

1. Finding Appropriate Objects – helping to decomposing system into object and identify less obvious abstractions and objects that represent process or algorithm that do not occur in real world,
2. Determining Object Granularity - object can represent whole system or subsystem or simple parts. Design patterns helps to decide what should be an object and design patterns such as Façade, Flyweight, AbstractFactory,and Builder, Visitor and Command pattern address this issue,
3. Specifying Object Interfaces – objects in object-oriented systems are only accessible through their interfaces. The objects with completely different implementation can share the same interfaces. An object can have many types and different objects can share the same type. Resolve to which object associate request during run-time is called dynamic binding. Dynamic binding allows substitution of objects with identical interfaces and this key concept in object-oriented systems is known as polymorphism, Design patterns helps to identify key elements of interfaces and data that should be exchanged as well as relationship between interfaces,
4. Specifying Object Implementations – an object implementation is specified by its class and class specifies object internal data and operations. Object is created from its class and supports his class interfaces. Class inheritance is mechanism for extending class functionality by reusing functionality in the parent class. Interface inheritance is mechanism for replacing an object use with another which implements the same interface and is considered to belong to common type. Many design patterns must have common type. Examples are ChainOfResponsability, Composite, Command, Observer, State and Strategy patterns,
5. Programming to an Interface, not an Implementation – polymorphism and reusability depends of the inheritance ability to define a set of object with identical interfaces. When request in object-oriented system is sent it does not ask for object type. Any object type that implements identical interfaces can process request. In-

terfaces are usually defined by abstract class which can specify purely virtual interfaces. Most important benefits are that client is not aware of object type as long as it implements interface type and client is unaware about classes that implements these objects (Gamma et al. 1994:30). Design patterns principle that reduces implementation dependencies between subsystems is **Program to an interface, not an implementation**! Commit only to an interface defined by an abstract class, but not to variables that are instances of concrete class. To instantiate concrete class which implements functionality design patterns offers creational patterns, Abstract Factory, Builder, Factory Method , Prototype and Singleton pattern,

6. Putting Reuse Mechanisms to Work – the real challenge is to build flexible and reusable software. In object-oriented systems reusability is achieved through class inheritance and object composition. White box is called reusability by sub classing because internal details are visible to subclass and black box is called reusability by composition because internal details are not visible. Both mechanisms have advantages and disadvantages. Inheritance breaks important object-oriented principle called encapsulation. Changes in the parent class would affect subclasses too. This kind of dependency limits fleksibility and reusability. One solution is to inherit only from abstract classes because there is no or very little of implementation. Object composition is preferred approach because composition is defined dynamically at run-time, so **Favor object composition over class inheritance** (Gamma et al., 1994).
7. Delegation – this technique makes composition as powerful as inheritance. Basically receiver passes itself to the delegate and delegate operation uses receiver operations. Composition disadvantages are run-time inefficiency and time necessary to understand dynamic and parameterized solution. Design Patterns that uses delegation are State, Strategy and Visitor. “*Delegation is an extreme example of object composition. It shows that you can always replace inheritance with object composition as a mechanism for code reuse*.” (Gamma et al., 1994, p. 34)
8. Inheritance versus Parameterized Types – even this is not strictly object-oriented technique, it lets you to provide type as an parameter when it is needed, Design patterns in this book do not use this technique.
9. Relating Run-Time and Compile-Time Structures – compile-time and run-time structures in object-oriented systems are quite different. Compile-time structure contains code and is frozen, but run-time structures are in constant changes and communication. Design patterns that captures differences between compile-time and run-time structures are Composite, Decorator, Observer and ChainOfResponsability.
10. Designing for Change –when designing system it must be considered how system could be changed during lifetime. The system must be robust enough to accept changes and modifications. Design patterns provides solution for creating object by specifying class explicitly, dependencies on specific operations, dependencies on software and hardware platform, dependencies on object representation or implementation, algorithmic dependencies, tight coupling, extending functionality by subclassing, inability to alter classes conveniently. These are examples that illustrate design patterns flexibility and ability to provide solution to design issues.

The second chapter is a case-study about designing document editor.

This popular book contains twenty three patterns divided in the three groups:

1. Creational Patterns,
2. Structural Patterns,
3. Behavioral Patterns.

The patterns examples in this book are presented by using C++ language.

These patterns are described in the Chapters 3, 4 and 5 and these chapters serve as "pattern catalogue" (Gamma et al., 1994).

> "*None of the design patterns in this book describes new or unproven designs. We have included only designs that have been applied more than once in different systems. Most of these designs have never been documented before. They are either part of the folklore of the object-oriented community or are elements of some successful object-oriented systems—neither of which is easy for novice designers to learn from.*" (p. 12)

Why this book becomes so popular? Answer to this question is not easy, but we can analyze facts that precede book publishing as well as facts that were following after this book publishing. One of the critical preceding factors was joining group of computer experts and consultants with impressive records and experience who become mutually interested to the promoting design patterns technique.

The names such as Grady Booch, the best known as, (together with Ivar Jacobsen and James Rumbaugh), one of the Unified Modeling Language (UML) creator, and chief scientist at Rational Software Corporation, Kent Beck, creator of Extreme Programming and Test Driven Development software development methodology, Ward Cunningham, who developed first Wiki, Doug Lea, a professor of computer science at State University of New York at Oswego, Bruce Andersen professor of computer science at University of Essex, James O. Coplien a research scientist at Bell Labs, Richard Gabriel .an expert on the Lisp programming language were guarantee that Design Patterns are emerging methodology that would designate coming years.

The publishing timing of the book, just before the first PLoP conference, was perfect. According to Addison-Wesley data, "*It sold 750 copies at the conference - more than seven times the highest number of any technical book Addison-Wesley had ever sold at a conference.*" (HistoryOfPatterns 2010).

This book presented design solutions that have been approved and signed by respectable amount of software engineering experts. Even it can be argue that it is not easy to grasp design patterns ideas from first reading; this book is still used as a reference and as helpful guide to the pattern catalogue.

Design Patterns (R)Evolution

Design Patterns were supported by large group of software engineering experts after the first PLoP conference in 1994 and huge number of books have been published and recommended using of the design patterns as best practice for object-oriented systems.

The five Siemens engineers, called Gang of Five (GoV) wrote "Pattern-Oriented Software Architecture: A System of Partners Volume 1" (Buschmann et al., 1996). This book is well known as POSA. Selected papers from the first and second PLoP and patterns Design (PLoD) conferences were published in the "Pattern Languages of Program Design" and "Pattern Languages of Program Design 2" books. These books and many other are part of Addison-Wesley Design Patterns series. Martin Fowler wrote "Patterns of Enterprise Application Architecture". Addison-Wesley published this book too (Fowler, 2003). On the HillSide Internet page at http://hillside.net/patterns/books is maintained catalogue of the books on patterns.

Since 1994 number of design patterns exploded and today we can count a hundred of patterns. There are many repositories which maintain and document patterns. One of the first established in 1995 is called “Portland Pattern Repository” and is maintain by Ward Cunningham.

Design patterns are enforcing object-oriented system core principles:

1. Encapsulation – data and implementation hiding,
2. Inheritance – a new class is created from existing one and inherits all attributes and behavior of parent class. This lind of relation is called child-Parent relationship or “is a” relationship
3. Polymorphism – ability of different object types to process call to the method of the same name.

The primary goal of design patterns is to promote proven design and repeatedly implement the successful solution template in the different context.

> “*A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context”. (Gamma et al., 1994, p. 399)*

Recognizing and adapting solution to different context is a challenge and a part of design pattern learning curve. The best way to collect experience and build knowledge about design patterns is through work done on the real world projects.

Martin Fowler in his book (2003) says about patterns:

> *“Patterns aren’t original ideas; they’re very much observations of what happens in the field. As a result, we pattern authors don’t say we ‘invented’ a pattern but rather that we ‘discovered’ one. Our role is to note the common solution, look for its core, and then write down the resulting pattern.*” (p. 10)

Martin Fowler (2003) divides patterns in the three groups, depending of the place in the layered architecture where pattern can be used:

1. Presentation layer,
2. Domain layer,
3. Data Source layer.

About patterns that are described in this book Martin Fowler says:

> “*As you consider using the patterns never forget that they’re a starting point, not a final destination. There’s no way that any author can see all the many variations that software projects have. I’ve written these patterns to help provide a beginning, so you can read about lesson that I, and the people I’ve observed, have learned from doing and struggling. You’ll have your own struggles on top of these. Always remember that every pattern is incomplete and that you have responsibility, and the fun, of completing it in the context of your own system.*” (2003, p. 13)

It is important to remember that patterns cannot be implemented blindly. Patterns are templates and should be adapted to the particular solution or context in which patterns are used.

Design will not very much benefit from independent pattern components which are used to build an application if these components are built without considering surrounding patterns. Even there can be local benefits, it would create complex architecture and fail to satisfy functional and quality requirements (Buschmann & Henney, 2008). Implementing patterns does not automatic leads to good design. Failing to consider pattern interaction and collaboration, as well as context in which are implemented, can lead to bad design

> *"But: applying patterns in software design is not necessarily designing with patterns!"* ((Buschmann & Henney, 2008)

According to the same authors and papers (Buschmann & Henney, 2008) good design with patterns needs to consider some fundamental principles:

1. ***Patterns integration -*** through ***refinement***, which *refines* a structure and a behavior of another patter and ***combination***, which combine two or more patterns to solve more complex problems ,
2. ***Patterns relationship regarding choices*** – through ***pattern alternatives***, which address choice issues related to the patterns that solves he same or similar issues but each pattern is slightly different and has different consequences and ***cooperation***, because some patterns cooperate and mutually reinforcing,
3. Applying patterns by considering their ***relationship*** to create ***balanced design***.

GoV (Buschmann et al., 1996) divides patterns to the:

1. Architectural patterns – high level view
2. Design Patterns – middle level design component,
3. Idioms – internal component design.

Design patterns foundation today is very huge and patterns are available everywhere. Patterns are used to the system decomposition, to defining and implementing system components, and describing communication between components. Different levels of patterns are used to the component internal design and implementation.

Higher level of components, such as architectural patterns, for example, can overlap by already existing technologies and methods for describing the same concept. Example is a layered and n-tier architecture where missing knowledge can lead to confusion and wrong design decision.

For example Model View Controller (MVC) pattern can looks very much to the layered architecture presented by Presentation, Application and Data layer. But there are fundamental differences because in three tier architecture client tier never communicate directly to the data tier (MultitierArchitectureWikipedia, 2010). In a simple implementation the MVC design pattern view is updated directly from a model. In a complex case there can be business and data components between and communication to the data tier would go through all these components.

Design Patterns is a concept that has own set of rules of engagement and own terminology that is called Patterns Language. Design Patterns are communicating with each other and particular pattern implementation can require or dependent of implementation of another pattern or patterns. Each pattern describes solution and consequences as well as context in which has been used.

Different authors describe pattern structure differently. Some authors are using a classic form description based on GoF or POSA books when describing pattern structure (Fowler, 1999). Martin Fowlers believes that classic pattern structure description defined by GoF (Gamma et al., 1994) is too small and he offers own pattern structure elements. According to Fowler (1999) pattern description should contain:

1. Name of pattern – this element is crucial because patterns are creating vocabulary which enables effective communication,
2. The intent and the sketch – the intent sums up pattern in few sentences and the sketch is visual representation, often but not always a UML diagram,
3. Motivating problem – one of solved problems that best motivates the pattern,
4. How it works – describes solution and implementation issues,
5. When it is used – describes when the pattern should be used and trade off, when there are pattern alternatives,
6. Further reading – points to reference bibliography and other pattern discussions,
7. Examples – one or more examples, illustrated with code.

Design patterns are not language specific. Design Patterns are describing solution to the common design problems.

Design patterns are reusable. Reusability means that object can be used and reused again and again. In this case it is mostly related to the control structures. However, implementation depends of required changes and very often shall be more or less replaced by new code or adapts existing code to the particular context.

Design pattern claims that solution is best practice and is most flexible. Flexibility means that it is possible easy replace implementation and refactoring of existing code and adaption to the required changes.

Design patterns are enforcing encapsulation. Encapsulation means that object inner implementation is hidden from the outside world and all communications with object are done through well defined interfaces. It means also that inner implementation can be replaced without affecting interface.

These who are familiar with patterns, developed Patterns Language where design problem is described by using involved pattern names The Abstract Factories and Builder patterns for creating classes, as well as Front Controllers or MVC pattern are becoming very often used as part of the everyday language to describe implemented or desired solution.

> *"Pattern names are crucial, because part of the purpose of patterns is to create a vocabulary that allows designers to communicate more effectively. Thus, if I tell you my Web server is built around a Front Controller and a Transform View and you know these patterns, you have a very clear idea of my web server's architecture."* (Fowler 2003)

Sun Microsystems Press and Prentice Hall made available the entire Enterprise Edition J2EE patterns catalog from the book "Core J2EE Patterns Best Practices and Design Strategies" 2nd and 1st edition at the http://www.corej2eepatterns.com/index.htm and http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html Internet sites.

After SUN being sold to Oracle, these pages become also Oracle SUN Developer Network (Oracle SUN Developer Network 2010). These sites provide access to J2EE patterns description and example code by clicking on pattern name. Click will lead to the page where each pattern is described and provided an example of Java language code. The 1st edition contains 15 patterns and 2nd edition contains 21 patterns.

Oracle SUN describes pattern structure by:

1. Pattern name,

2. Context - layer in which pattern is used and general description of processing requirements and pattern purpose,
3. Problem – specific problem that particular pattern should solve,
4. Forces – specific requirements within context that particular pattern solves, often supported by an practical example,
5. Solution – detailed description when pattern should be used and eventually interaction or combination with other patterns,
6. Structure – UML class diagram of particular pattern,
7. Participants and Responsabilities – UML sequence diagram which is representing pattern and description of each participants,
8. Strategies – implementation details and recommendations as well as example code and UML diagrams when it is necessary to explain more details,
9. Consequences – short overview of advantages of implementing pattern, but avoiding to mention explicitly drawbacks,
10. Related Patterns - list of patterns which can be combined with pattern or list of alternative patterns and patterns combination.

MSDN Microsoft provides "Enterprise Solution Patterns Using Microsoft .NET" Internet page at http://msdn.microsoft.com/en-us/library/ff647095.aspx and "patterns & practices" Internet page at http://msdn.microsoft.com/en-us/practices/default.aspx. On this Internet pages are described architecture and enterprise patterns, as well as it is presented example code. In the book "Enterprise Solution Patterns Using Microsoft .NET 2.0" (Trowbridge, 2003) patterns are described by:

1. Pattern name,
2. Context - layer in which pattern is used and general description of processing requirements and pattern purpose,
3. Forces – consideration within particular context ,regarding solution selection,
4. Solution – solution description, class diagram and short description of responsibility of each involved class module,
5. Variations – presents variations of particular implementation and discuss each particular solution and provides an UML class diagram,
6. Example – reference to an example,
7. Testing consideration – testability possibilities regarding to particular solution
8. Resulting Context – which describes **Benefits** and **Liabilities,**
9. Variants – specifics related to the pattern implementation,
10. Related patterns – reference to related patterns,
11. Acknowledgements – references to those who are beginning development of particular pattern and reference to used literature.

Foreword for this book has been written by Ward Cunningham, He wrote:

> "*Whenever we pull patterns together our choices say something important about how we work. Our philosophy of work runs through our selections. For example, in the Design Patterns book, [Gamma, et. al, Addison-Wesley], the philosophy was to make programs flexi-*

ble. This is important, of course, and some of those patterns are included here. But there are two other philosophies present in this volume worth mentioning. One philosophy is that in a continuously evolving environment like the enterprise, every complexity has a cost.

Another philosophy that runs through these patterns is that different people in the enterprise use different patterns for different purposes. Some patterns are more about the user experience than anything else“. (Trowbridge et al., 2003, p. 51)

Anti-patterns

“*Anti-pattern is just like pattern, except that instead of solution it gives something that looks superficially like a solution, but isn't one.*" (Koenig, 1995)

Anti-pattern term has been created by Andrew Koenig in his article “Patterns and AntiPatterns”, published by Journal of Object-Oriented Programming in 1995.

Anti-pattern is opposite from pattern, and is used to describe an implementation that even commonly used, in practice can be ineffective and produce more issues than benefits. Such implementation needs to be refactored.to become an effective solution.

Brown et al. (1998) distinguish anti-patterns and bad-practice or bad solution. AntiPattern is (Wikipedia-Anti-pattern, 2010):

- *Some repeated pattern of action, process or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results, and*
- *A refactored solution exists that is clearly documented, proven in actual practice and repeatable,*

While design patterns are offering unique solutions for known object-oriented design problems, AntiPatterns are offering alternative solutions and awareness of situation. (Brown et al., 1998) The careless changes can cause that effective solution becomes ineffective and change implemented pattern to anti-pattern. Anti-patterns become widely popular after anti-pattern book (Brown et al., 1998) that beside software patterns include also social anti-patterns.

Authors came with own Anti-Pattern definition (Brown et al., 1998):

- *AntiPatterns are Negative Solutions that presents more problems than they address,*
- *AntiPatterns are natural extension to design patterns,*
- *AntiPatterns bridge the gap between architectural concept and real-world implementations,*
- *Understanding AntiPatterns provides the knowledge to prevent or recover from them.*

The AntiPatterns in this book are divided in the three groups (Brown et al., 1998; Wikipedia-Anti-pattern, 2010) according to following viewpoints:

1. Development AntiPatterns,
2. Architecture AntiPatterns,
3. Management AntiPatterns.

Each of these groups identifies numbers of AntiPatterns and contains additional group of Mini anti-patterns. For example Development AntiPatterns group contains following anti-patterns (Brown et al., 1998):

1. Lava Flow – undocumented, redundant or undesired piece of code that cannot be easy refactored or removed,
2. Poltergeist (Proliferation of Classes) – an object that suddenly starts to execute functionality and then disappear. Called also a Gypsy Wagon,
3. Spaghetti Code – code that is missing structure or is not designed according to object-oriented design practice,
4. The Blob (God Class) – single class with too many attributes and operations or collection of unrelated attributes and operations that is not easy to modify without affecting overall functionality ,
5. Cut and Paste Programming – copy and paste instead to make general solution,
6. Functional Decomposition – ignoring object-oriented design and designing each function as a class,
7. Golden Hammer – implementing the same solution everywhere..

And Development Mini-AntiPatterns group contains following patterns (Brown et al., 1998; Wikipedia-Anti-pattern, 2010):

1. Ambiguous Viewpoint - a model without viewpoint,
2. Boat Anchor – missing to remove a part of a system that is not use anymore,
3. Continuous Obsolescence – difficulties to stay up to date with continuosly changing technologies,
4. Dead End – modification of commercial software creates maintenance burden,
5. Input Kludge - users and testers can easily discover too many bugs,
6. Mushroom Management – keep developer in the dark (keeping developers uninformed).

The Mini-AntiPatterns represents AntiPatterns that are easy to understand and can be described by using mini AntiPattern template that contains AntiPattern Name, AntiPattern Problem and Refactored Solution.

> *"The full AntiPattern template provides a wide range of information about a particular class of problems including a discussion of the AntiPattern background, the general form, symptoms and consequences, root causes, a refactored solution, and an example detailing how the refactoring process can be applied to create an effective solution."* (Brown et al., 1998)

The Wikipedia (Wikipedia-Anti-pattern, 2010) contains list of Known anti-patterns, divided into:

1. Organizational anti-patterns
2. Project management anti-patterns
3. Analysis anti-patterns
4. Software design anti-patterns
5. Object-oriented design anti-patterns

6. Programming anti-patterns
7. Methodological anti-patterns
8. Configuration management anti-patterns

The Wikipedia anti-pattern groups are different than viewpoints used by Brown et al. (1998). An example is anti-pattern called Mushroom Management classified as Development AntiPatterns (Brown et al., 1998) and as Organizational anti-pattern at Wikipedia (Wikipedia-Anti-pattern, 2010). The Wikipedia title "Known anti-patterns" is ambitious but it does not contain all patterns mentioned in the Brown et al. book (Brown et al. 1998).

Why it is important to know about AntiPatterns?

The AntiPatterns main focuses are bad solutions and solutions that should be avoid or at least warn that particular solution is potentially dangerous and could more or less jeopardize project success and cause profit loss. The Anti-Patterns, as well as Design Patterns are creating common vocabulary. The AntiPatterns also makes aware that software is subject of changes and each change, if not properly handled, can cause further troubles.

"*Yesterday's hot solution can become today's AntiPattern*". (Brown et al., 1998)

Andrew Koenig offers his own answer to question, Why it is important to know about AntiPatterns?

> *"If one does not know how to solve a problem, it may nevertheless be usefull to know about likely blind alleys. This is particularly true when something appears to be a solution but further analyses proves it is not. Even if one knows right answer, however, it may be important to point out particular hazard associated with that answer or seemingly trivial variations of that answer that turn solutions into non-solutions*". (Koenig, 1995)

General Discussion

There is no doubt that Design Patterns have positively influenced software engineering in last decade. Design Patterns becomes common repository where has been recorded software engineering knowledge and experience that is widely accepted as best practice. Until now this fact is widely accepted as truth in case of Software Design Patterns. Design Patterns provided a common mechanism for sharing knowledge and a experience and created global discussion about what best practice is?

The first what has to be noticed is that design patterns are templates. These templates shall be adapted to particular context.

There is not available only single solution to a particular problem. Rather are offered solutions that can satisfy different level of complexity. To solve simple problem are offered simple patterns. If complexity increases, it requires implementation of different patterns and refactoring of existing code.

Can we conclude that it is best practice use most complex patterns solution from project start and avoid expensive refactoring and recoding? That would be a wrong conclusion.

"*A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system.*" (John Gall)

Developer needs more time to understand complex design and also more time to implement it, but projects are always limited by time constraints and financing constraints, as well as with available resources, human resources, hardware and infrastructure, and also limited by limitations of available technology, as for example wireless network bandwidth.

"*One philosophy is that in a continuously evolving environment like the enterprise, every complexity has a cost. You'll find a variety of patterns here that at first seem contradictory. That's because the authors know that successful enterprise applications start simple and grow over time. Something simple works for a while then it needs to be replaced.*" (Ward Cunningham of Cunningham & Cunningham, Inc, Foreward to"Microsoft Enterprise Solution Patterns Using Microsoft.NET".)

Other authors are also recommending avoiding complexity when it is possible.

"Adding complexity obscures your intentions, making the system more difficult for other developers to understand. The added complexity also makes the system harder to maintain and extend, thereby increasing the total cost of ownership. If this added complexity is carefully considered and reserved for meeting current requirements, it can be worthwhile. Extra complexity is sometimes added based on speculation that it might be needed someday, rather than based on current requirements. This can clutter code with unnecessary abstractions that impede understanding and your ability to deliver a working system today." (Trowbridge et al., 2003, p. 51)

Increased complexity in case of using Design Patterns is paid by flexibility and easy maintenance according to Design patterns experts. But there are also present other arguments that say that it takes longer to understand source code that implements Design Patterns. The simple reason is that generic and parameterized solutions are more complex and not easy understandable.

There is also another well know issue in case when there are available number of different solution. If there is more solutions and solutions variants available then it will require longer time to choose solution. If there is more combination then it will take longer time to compose desired solution. It will take even more time to test each of solution and different combinations and variations.

"*A key part of patterns is that they're rooted in practice. You find patterns by looking at what people do, observing things that work, and then looking for the 'core of solution'. It isn't an easy process, but once you've found some good patterns they become a valuable thing.*" (Fowler, 1999)

This means also that overall project cost will increase.

"*Unfortunately, there is no single design strategy that is right for all situations. This is due to the competing needs in software design to eliminate excessive redundancy and excessive complexity.*" (Trowbridge et al., 2003, p. 51)

Design Pattern implementation can be vendor specific. One example is Microsoft implementation of Model View Controller (MVC) pattern. In the Microsoft MVC pattern implementation, the Controller is merged into a View and Observer pattern is used as notification mechanism when Model is changed. This MVC implementation is called the Document-View variant.

There are also existing Architectural Design Patterns and three layer architecture is widely accepted as best practice. Distributed application architecture is a set of the interacting components. Each component encapsulates functionality and exposes public interface for interaction with another components. As long as component provides required functionality, the internal component implementation is not important. The important part is what data need to be sent to component and what response will be received from the component (Microsoft, 2002, p. 5).

"By identifying the generic kinds of components that exists in most solutions, you can construct a meaningful map of an application or service, and then use this map as a blueprint for your design." (Microsoft, 2002, p. 7).

The components that provide similar functionality can be grouped into layers (Microsoft, 2002, p. 5). By identifying common application layers can be created reusable Architecture Pattern. Presentation Layer, Business Layer and Data Layer are recommended by followers and supporters of each of the leading Enterprise technologies vendors. Differences are more obvious in the domain of the implemented technologies, tools and utilities that are available to support reference implementation in each of these layers.

Learn and understand design patterns needs time and most important part of this process is implementing design patterns in the real world projects. Choosing and implementing Design Patterns first time is a challenge.

Development is an iterative process and implementing Design Patterns do not change this fact. Each iteration should improve design and implementation as long until it satisfies software requirements. The basic Design Pattern idea is to "programming to an Interface, not an Implementation" (Gamma et al. 1994, p. 30).

The most important advantages of programming to interface are (Gamma et al., 1994, p. 30):

1. Client remains unaware of specific type of object they use,
2. Client remains unaware of the classes that implement these objects.

In both of above cases client knows only about abstract class that defines interface.

Long running software can introduce unforeseen issues that can invalidate more and less abstract solution. As it is today, deployment and testing can give better picture about system functionality and finally only a system running in production over longer time can confirm if it can satisfy functionality, reliability, scalability and performances, and flexibility requirements. Changes of requirements and changes of project resources can create serious challenges too.

Software design is under constant changes as well as software requirements are. It means that once selected pattern cannot expect to be used forever. Changed requirements can enforce using different pattern. This is today commonly referenced as code refactoring. Code refactoring means that classes and methods are organized differently to support reusability and improve code structure and maintainability.

Another issue when patterns are selected is the collaboration of different patterns. In the perfect world, design patterns should form independent components that are communicating to each others by exchanging messages, what is by the way idea behind object-oriented systems. But our world is imperfect. It means that some patterns are collaborating and even worst depending of the implementation of other patterns. This also means that choosing to use one kind of pattern would automatically lead to using particular pattern or set of patterns which would support implementation. This is especially truth in case of layered architecture where each layer contains set of patterns which role is well defined in advance.

"Each pattern is relatively independent, but patterns aren't isolated from each other. Often one pattern leads to another or one occurs only if another is around." (Fowler, 1999, p. 10)

The very important is to remember that patterns are templates. These are not solution that can be implemented directly. Patterns are an example of the similar problem and solution that has been implemented in particular context and observed, analyzed and understand. The understanding is based on the experience collected through the work done on the real project and issues that had been raised when solution has been developed and deployed, hopefully in real production environment. Each advantage and disadvantage has been described. This part of design patterns is very important because it can be used as inputs to risk management and prepare future strategy for mitigate possible issues.

Discussion about patterns cannot be completed without mentioning Anti patterns. Anti patterns are about bad design and bad practice that can occur also in case of constant changes and corrupt initially well designed and structured software. The changes are always depended of the experience and knowledge of the particular person who is implementing changes. To mitigate this issue is strongly recommended code review and frequent testing. Automated regression testing is in this case invaluable tool.

Communicating through interfaces is an old and well known issue and it exists much longer then object-oriented programming. The very basic issue is a change of component interface. In the complex system, where more than one system is using the same component, if interface is changed then all components that are using the same interface should be changed. This is not always possible because other systems or components can be owned by another system that is not interesting to make any changes. Even we own all components in case when different customers are running different system versions, some of them would require changes and some would not.

In this case interfaces might not be changed or in case that it has too, it is done through creating new interface that is a copy of an old interface and then changing this interface according to required changes. The old interface is still available and is not changed.

Another approach to solve this issue is component versioning. It means that during creating runtime modules are specified component versions that are supporting certain interfaces.

Both solutions are increasing maintenance complexity. Number of interfaces could grow during time and in case of versioning and creating new software version can be very complex task.

One of the goals of the Java object-oriented language was reducing number of classes. But praxes show that number of objects and methods is exploding and even simple application has hundreds of different classes. Design Patterns are increasing complexity and increasing number of object. Design patterns and refactoring increases number of classes and methods.

There is also important to know an answer to question “Can Design Patterns help to avoid faulty software which is not able to satisfy customers’ requirements?”

No they cannot!

Design Patterns are mostly related to the component internal design and interactions with other components, and assuming that software requirements are well known in advance. Specifying software requirements is a different subject.

Design Patterns Examples

Examples in this chapter are based on the Creational patterns (Gamma et al., 1994). Each Design pattern example contains fully working example written in Java programming language.

Gamma et al. (1998) described five Creational patterns;

1. Abstract Factory – provides interface for creating families of related or dependent objects,
2. Builder – separates construction of complex object from representation and use the same construction process to create different representations,
3. Factory Method – defines interface for creating an object, but subclasses decide which class to instantiate,
4. Prototype – creates new object by cloning old one and changing necessary attributes values,
5. Singleton – provides mechanism for instantiation of only one class instance and creates global point of access to it.

While it is easy to see differences between Prototype and Singleton and other Creational Patterns, Abstract Factory and Factory Method as well as Builder can make confusion to novice developer. The next section will use examples to illustrate Abstract factory design pattern.

Abstract Factory

Abstract Factory design pattern is described as an interface for creating families of related or dependent objects without specifying concrete class (Gamma et al., 1998). A client application in the Abstract Factory pattern using generic interfaces to access concrete class and is not aware of which object is using. Instantiation of concrete class is hidden inside of the Factory.

A Factory in Design Patterns vocabulary describes a place where other objects are constructed. A Factory usually lets subclasses decide about which class should be instantiated and this technique is also known as deferred instantiation. The following classes are involved in designing of Abstract Factory pattern:

1. AbstractFactory - declares an interface for operations that create abstract product objects.
2. ConcreteFactory - implements the operations to create concrete product objects.
3. AbstractProduct - declares an interface for a type of product object.
4. ConcreteProduct - defines a product object to be created by the corresponding concretefactory.implements the AbstractProduct interface.
5. Client - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Examples in the following sections are based on a fictive requirement to provide menu list for different restaurants. A menu list can be used, for example, to decide in which kind of restaurant one should have a dinner or make table reservation.

Focus in the two first implemented versions is on the restaurant type and corresponding menu list. For example an Italian restaurant shall display a menu list of Italian meals and an Indian restaurant shall display a menu list that contains Indian meals.

The second implemented version demonstrate how easy is to add new restaurant type to existing code.

The third implementation version extending this example by another fictive requirement to provide a list of vendors which are able to deliver furniture that fits to the restaurant type. For example Italian restaurant shall use Italian furniture. This example demonstrate how much work is necessary to extend existing example.

Abstract factory example v1

The application client in this example creates different types of restaurants and displays menu list that corresponds to restaurant type. This version implements Italian restaurant and Indian restaurants type.

The following diagram contains example structure:

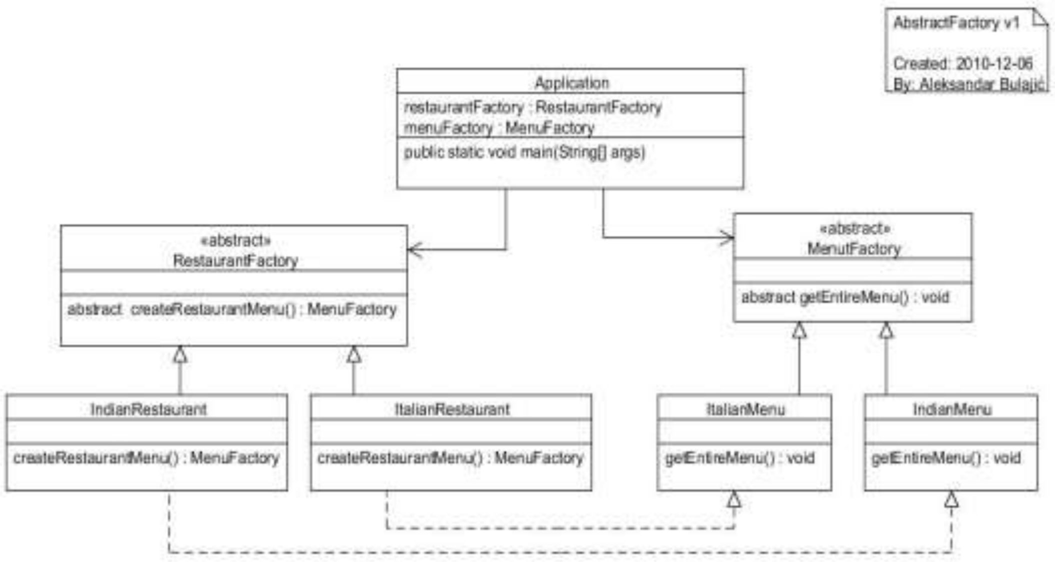
AbstractFactory v1
Created: 2010-12-06
By: Aleksandar Bulajic
Application
restaurantFactory : RestaurantFactory
menuFactory : MenuFactory
public static void main(String[] args)
«abstract» RestaurantFactory
abstract createRestaurantMenu() : MenuFactory
«abstract» MenutFactory
abstract getEntireMenu() : void
IndianRestaurant
createRestaurantMenu() : MenuFactory
ItalianRestaurant
createRestaurantMenu() : MenuFactory
ItalianMenu
getEntireMenu() : void
IndianMenu
getEntireMenu() : void

Figure 1 Abstract Factory Example v1

This simple structure satisfies requirements for Abstract Factory design patterns participants:

1. Abstract class **RestaurantFactory** is an **AbstractFactory** class,
2. Classes **IndianRestaurant** and **ItalianRestaurant** are both **ConcreteFactory**,
3. Abstract class **MenuFactory** is an **ProductFactory** class,
4. Classes **ItalianMenu** and **IndianMenu** are both **ConcreteProduct**,
5. **Application** class is **Client** class.

A client application class (`Application`) is using generic interface (`RestaurantFactory.createRestaurantMenu()`) to instantiate an abstract class (`MenuFactory`).

A client application class (`Application`) does not know which class exactly will be instantiated. Access to the concrete classes instantiation are hidden inside of the generic interfaces (`RestaurantFactory.createRestaurantMenu()` and `MenuFactory.getEntireMenu()`), that are in this case defined as two abstract methods inside of the abstract classes (`RestaurantFactory and MenuFactory`).

The following source code provides more details.

```
public abstract class RestaurantFactory {
    public abstract MenuFactory createRestaurantMenu();
}
```

The abstract class `RestaurantFactory` defines **`public abstract`** method `createRestaurantMenu()`. This method return type is a `MenuFactory` class type.

Note that `RestaurantFactory` class returns different class type MenuFactory. That is exactly primary role of `Factory class` in object-oriented system. Further, this class and method `createRestaurantMenu()` are declared abstract, what means that this method should be implemented inside of the each subclass.

This is exactly what concrete classes `IndianRestaurant` and `ItalianRestaurant` are doing. The next source code example illustrate this kind of implementation.

```
public class IndianRestaurant extends RestaurantFactory {
     @Override
     public MenuFactory createRestaurantMenu() {
          return new IndianMenu();
     }
}
public class ItalianRestaurant extends RestaurantFactory {
     @Override
     public MenuFactory createRestaurantMenu() {
          // TODO Auto-generated method stub
          return new ItalianMenu();
     }
}
```

Note in above example that even in both classes, `createRestaurantMenu()` method returns the same class type `MenuFactory,` there are returned different instances of concrete classes.

`IndianRestaurant.createRestaurantMenu()` returns instance of `IndianMenu` class and `ItalianRestaurant. createRestaurantMenu()` returns instance of `ItalianMenu` class.

Look at the Abstract Factory structure in Figure 1. The above examples are covering the left half part of the class diagram.

Now is time to create next abstract class `MenuFactory.`

```
public abstract class MenuFactory {
     public abstract void getEntireMenu();
}
```

On the first look this class is very similar to a `RestaurantFactory` class. But there are very important differences.

First one is that MenuFactory does not contain any kind of create method and does not returns another class type. `MenuFactory` contains set of implementation methods, that are implemented in each of `MenuFactory` subclasses, `ItalianMenu` class and `IndianMenu` class.

This of course does not mean that `MenuFactory` cannot define or implement other kind of methods. This only means that in this case has been decided to use this solution.

```
public class ItalianMenu extends MenuFactory {
     @Override
     public void getEntireMenu() {
          System.out.println ("##########################");
          System.out.println ("Italian Menu\n\n"
                    + "  ItalianAppetizer1\n"
                    + "  ItalianAppetizer2\n\n"
                    + "  ItalianMainCourse1\n"
                    + "  ItalianMainCourse2\n\n"
                    + "  ItalianDesert1\n"
                    + "  ItalianDesert2\n\n"
                    );
     }
}
public class IndianMenu extends MenuFactory {
     @Override
     public void getEntireMenu() {
          System.out.println ("##########################");
```

```
        System.out.println ("Indian Menu\n\n"
                            + "IndianAppetizer1\n"
                            + "IndianAppetizer2\n\n"
                            + "IndianMainCourse1\n"
                            + "IndianMainCourse2\n\n"
                            + "IndianDesert1\n"
                            + "IndianDesert2\n\n"
                            );
    }
}
```

Above two classes are implementing abstract method defined in the abstract class `MenuFactory` and displays menu list, what was original requirement of this example. This is an education and demonstration example and it is main reason why it is simple and why the methods are mostly empty.

The real world example will be more complex and require further model decomposition, as for example introducing different kind of menus according to the daylight changing, breakfast, lunch and dinner menus as well as creating menus from different meals combinations. This will require modeling of additional classes for menu type and meals and dishes, pricelist etc. This is out of scope of this article.

Now, when all required classes are ready, we are going to create `Application` client class. This class is important because it is responsibility of this class to use previously created classes to provide correct functionality.

```
public class Application {
    public enum RestaurantType {
        Indian,
        Italian
    }
    /**
     * @param args
     */
    public static void main(String[] args) {
        RestaurantFactory restaurantFactory = null;
        MenuFactory menuFactory = null;

        RestaurantType restaurantType = RestaurantType.Italian;

        switch (restaurantType) {
            case Indian:
             restaurantFactory = new IndianRestaurant();
             break;
            case Italian:
             restaurantFactory = new ItalianRestaurant();
             break;
        }

        menuFactory = restaurantFactory.createRestaurantMenu();
menuFactory.getEntireMenu();
    }
}
```

First what can be noticed is that this class contains more code than other classes. Reason for this is the education purpose and step by step demonstration. The most of this code can be refactored and divided in more methods or even moved to the already described abstract classes and concrete classes.

But most important is to notice how Abstract Factory pattern is using previously created classes.

The Abastract Factory pattern does not instantiate concrete classes. Instead it instantiate abstract classes:

```
RestaurantFactory restaurantFactory = null;
     MenuFactory menuFactory = null;
```

To the `restaurantFactory` is assigned proper restaurant by selsction through `restaurantType` enumerated value. In this case to this variable is assigned value `RestaurantType.Italian` what forced switch statement to assign `ItalianRestaurant` to the a`restaurantFactory`:

```
     restaurantFactory = new ItalianRestaurant();
```

The MenuFactory class is assigned to `menuFactory` by calling RestaurantFactory `restaurantFactory.createRestaurantMenu()` method. This is what Factory create method usually do:

```
 menuFactory = restaurantFactory.createRestaurantMenu();
```

The `restaurantFactory.createRestaurantMenu()` return type is an abstract class `MenuFactory` type. The last line is just displaying restaurant menu.

```
menuFactory.getEntireMenu();
```

When this code is executed it displays Italian restaurant menu list:

```
##########################
Italian Menu

  ItalianAppetizer1
  ItalianAppetizer2

  ItalianMainCourse1
  ItalianMainCourse2

  ItalianDesert1
  ItalianDesert2
```

In this example is important to understand role of Factory methods and role of abstract classes.

Factory method returns different instance of a class type.

Abstract class provides a level of indirection and hides from an application client class instantiation. Application client is not aware which class type is instantiated. Hiding class type instantiation and making decision during runtime is the primary purpose of this pattern. It provides significant level of flexibility and enables code changes without significantly affecting the rest of involved classes and protecting a client from detailed knowledge about each involved classes.

Application client is using abstract class interface to communicate with other classes presented in this model.

Abstract factory example v2

This example will demonstrate how easy is to add new restaurant type, for example a ChineseRestaurant type to existing model and simplicity of changes to the existing code. This example is based on previous example and here will be presented only classes where is necessary to change code.

The following class diagram illustrates model changes.

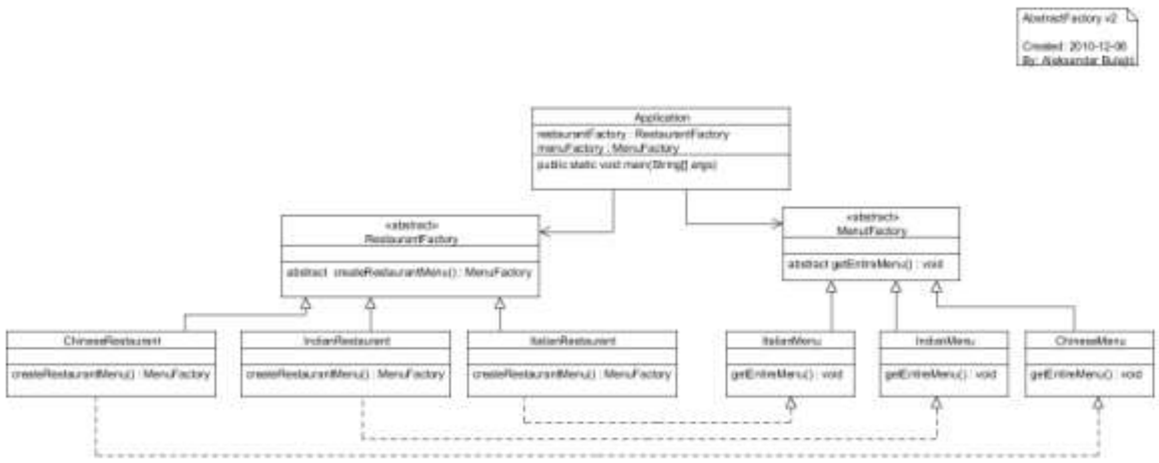
AbstractFactory v2
Created: 2010-12-06
By: Aleksandar Bulajic
Application
restaurantFactory : RestaurantFactory
menuFactory : MenuFactory
public static void main(String[] args)
«abstract»
RestaurantFactory
abstract createRestaurantMenu() : MenuFactory
«abstract»
MenuFactory
abstract getEntireMenu() : void
ChineseRestaurant
createRestaurantMenu() : MenuFactory
IndianRestaurant
createRestaurantMenu() : MenuFactory
ItalianRestaurant
createRestaurantMenu() : MenuFactory
ItalianMenu
getEntireMenu() : void
IndianMenu
getEntireMenu() : void
ChineseMenu
getEntireMenu() : void

Figure 2 Abstract Factory Example v2

Compared to class diagram in Figure 1, here are visible two new classes, `ChineseRestaurant` class and `ChineseMenu` class.

The `ChineseRestaurant` class is defined simple as subclass of the `RestaurantFactory` class:

```
public class ChineseRestaurant extends RestaurantFactory {
     @Override
     public MenuFactory createRestaurantMenu() {
          return new ChineseMenu();
     }
}
```

The class is defined as subclass of the `MenuFactory` class:

```
public class ChineseMenu extends MenuFactory {
     @Override
     public void getEntireMenu() {
          System.out.println ("##########################");
          System.out.println ("Chinese Menu\n\n"
                              + "ChineseAppetizer1\n"
                              + "ChineseAppetizer2\n\n"
                              + "ChineseMainCourse1\n"
                              + "ChineseMainCourse2\n\n"
                              + "ChineseDesert1\n"
                              + "ChineseDesert2\n\n"
                              );
     }
}
```

The both of above classes are brand new. Now we are going to make changes in the Application client class Application. These changes are very simple and limited to adding new enumeration type *`Chinese`*:

```
public enum RestaurantType {
        Indian,
        Italian,
        Chinese
}
```

and new case in the switch statement

```
case Chinese:
                restaurantFactory = new ChineseRestaurant();
                break;
```

This is how the `Application` class looks after these changes:

```
public class Application {
     public enum RestaurantType {
        Indian,
        Italian,
        Chinese
     }
     public static void main(String[] args) {

          RestaurantFactory restaurantFactory = null;
          MenuFactory menuFactory = null;
          FurnitureFactory furnitureFactory = null;

          RestaurantType restaurantType = RestaurantType. Chinese;

          switch (restaurantType) {
              case Indian:
                restaurantFactory = new IndianRestaurant();
                break;
              case Italian:
                restaurantFactory = new ItalianRestaurant();
                break;
              case Chinese:
                restaurantFactory = new ChineseRestaurant();
                break;
          }
          menuFactory = restaurantFactory.createRestaurantMenu();
          menuFactory.getEntireMenu();
}
```

When this code is executed it displays Chinese restaurant menu list because restaurantType is set to Chinese restaurant type (`RestaurantType restaurantType = RestaurantType. Chinese;`):

```
##########################
Chinese Menu

ChineseAppetizer1
ChineseAppetizer2

ChineseMainCourse1
ChineseMainCourse2
```

First of all note that there is not necessary to make any code changes in the abstract class `Res-taurantFactory` and in the abstract class `MenuFactory`.

Next what is important is that changes in the Application client are very simple and in this example can be done almost automatically by adding new enumeration type and change switch statement.

The next example will show more complex changes in case when it is necessary to add new product.

Abstract factory example v3

The last Abstract Factory example will demonstrate how much of additional work is necessary to extend existing code with another product, for example furniture vendor.

A furniture vendor shall of course correspond to restaurant type. This means that Italian restaurant shall be equipped by Italian furniture and Chinese restaurant shall be equipped by Chinese furniture.

The following class diagram illustrates the full class model.

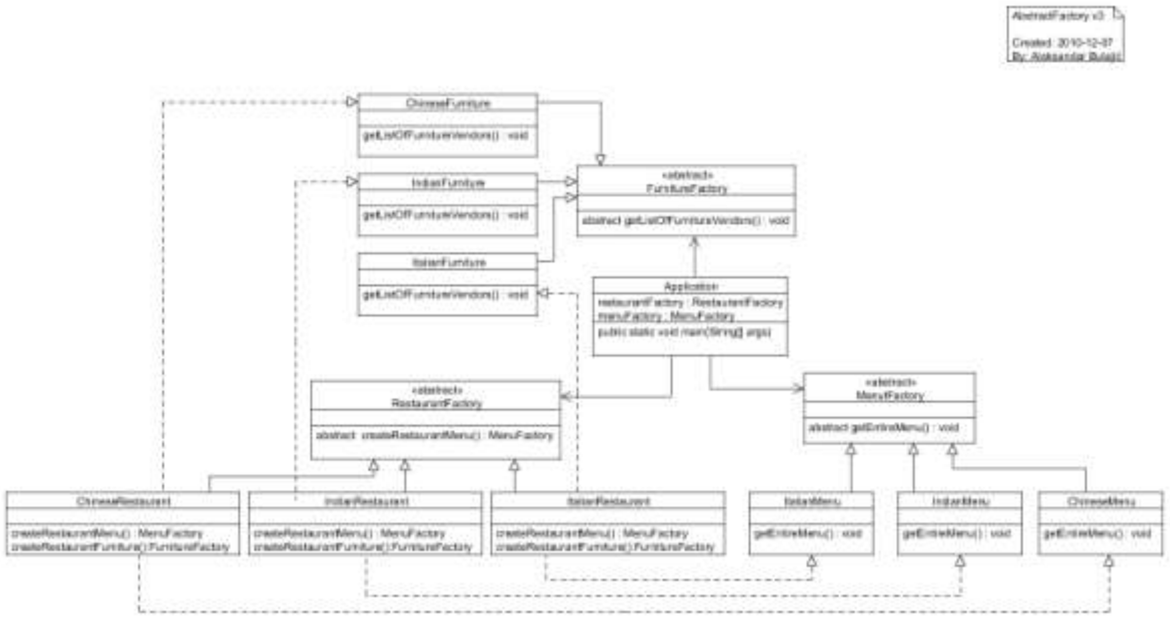
AbstractFactory v3
Created: 2010-12-07
By: Aleksandar Bulajic
ChineseFurniture
getListOfFurnitureVendors() : void
IndianFurniture
getListOfFurnitureVendors() : void
ItalianFurniture
getListOfFurnitureVendors() : void
«abstract»
FurnitureFactory
abstract getListOfFurnitureVendors() : void
Application
restaurantFactory : RestaurantFactory
menuFactory : MenuFactory
public static void main(String[] args)
«abstract»
RestaurantFactory
abstract createRestaurantMenu() : MenuFactory
«abstract»
MenuFactory
abstract getEntireMenu() : void
ChineseRestaurant
createRestaurantMenu() : MenuFactory
createRestaurantFurniture():FurnitureFactory
IndianRestaurant
createRestaurantMenu() : MenuFactory
createRestaurantFurniture():FurnitureFactory
ItalianRestaurant
createRestaurantMenu() : MenuFactory
createRestaurantFurniture():FurnitureFactory
ItalianMenu
getEntireMenu() : void
IndianMenu
getEntireMenu() : void
ChineseMenu
getEntireMenu() : void

Figure 3 Abstract Factory Example v3

This class diagram contains new abstract class *FurnitureFactory* and also three concrete subclasses, *ChineseFurniture*, *IndianFurniture* and *ItalianFurniture*.

The abstract class `FurnitureFactory` is a new Factory class. This class defines abstract method `getListOfFurnitureVendors()`:

```
public abstract class FurnitureFactory {
     public abstract void getListOfFurnitureVendors();
}
```

Each of three new subclasses provides implementation of abstract method `getListOfFurnitureVendors()`:

```
public class ChineseFurniture extends FurnitureFactory {
     @Override
     public void getListOfFurnitureVendors() {
          System.out.println ("##########################");
          System.out.println ("Chinese Furniture Vendors");
          System.out.println ("  Vendor 1");
          System.out.println ("  Vendor 2");
          System.out.println ("  Vendor 3");
     }

}
public class IndianFurniture extends FurnitureFactory {
     @Override
     public void getListOfFurnitureVendors() {
          System.out.println ("#########################");
          System.out.println ("Indian Furniture Vendors");
          System.out.println ("  Vendor 1");
          System.out.println ("  Vendor 2");
          System.out.println ("  Vendor 3");
     }

}
public class ItalianFurniture extends FurnitureFactory {
     @Override
     public void getListOfFurnitureVendors() {
          System.out.println ("#########################");
          System.out.println ("Italian Furniture Vendors");
          System.out.println ("  Vendor 1");
          System.out.println ("  Vendor 2");
          System.out.println ("  Vendor 3");
     }

}
```

All above are new classes.

Now is necessary to make changes in the existing code to implement new functionality. In this particularly example it should provide list of furniture vendors for furniture that corresponds to the restaurant type.

First will be changed `RestaurantFactory` class by adding new abstract method `createRestaurantFurniture()` which returns result of abstract class `FurnitureFactory`.

Here is the full class. The new added method is highlited:

```
public abstract class RestaurantFactory {
     public abstract MenuFactory createRestaurantMenu();
     public abstract FurnitureFactory createRestaurantFurniture();
}
```

When new abstract method is added to an abstract class, then all corresponding subclasses shall be changed and provide implementation of new abstract method. This means that all three subclasses, `ChineseRestaurant`, and `IndianRestaurant`, and `ItalianRestaurant` has to provide implementation of new method `createRestaurantFurniture()`. Here is the full class code and changes are highlighted:

```
public class ChineseRestaurant extends RestaurantFactory {
     @Override
     public MenuFactory createRestaurantMenu() {
          return new ChineseMenu();
     }
     @Override
     public FurnitureFactory createRestaurantFurniture() {
          return new ChineseFurniture();
     }
}
public class IndianRestaurant extends RestaurantFactory {
     @Override
     public MenuFactory createRestaurantMenu() {
          return new IndianMenu();
     }

     @Override
     public FurnitureFactory createRestaurantFurniture() {
          return new IndianFurniture();
     }
}
public class ItalianRestaurant extends RestaurantFactory {

     @Override
     public MenuFactory createRestaurantMenu() {
          // TODO Auto-generated method stub
          return new ItalianMenu();
     }
     @Override
     public FurnitureFactory createRestaurantFurniture() {
          return new ItalianFurniture();
     }
}
```

The last step is to change `Application` class and implement new functionality added to `RestaurantFactory` abstract class. This is done by following three lines of code:

```
FurnitureFactory furnitureFactory = null;
furnitureFactory = restaurantFactory.createRestaurantFurniture();
furnitureFactory.getListOfFurnitureVendors();
```

The first line declares `furnitureFactory` abstract class.

The second line initiates `furnitureFactory` by calling `RestaurantFactory` class `restaurantFactory.createRestaurantFurniture()` method which would return back instance of concrete `FurnitureFactory class.`

The third line is simple executing method from a concrete class to display furniture vendor list.

Following is the full class source code and lines where changes applied are highlighted:

```
public class Application {
     public enum RestaurantType {
         Indian,
         Italian,
         Chinese
     }
     /**
      * @param args
      */
     public static void main(String[] args) {

          RestaurantFactory restaurantFactory = null;
          MenuFactory menuFactory = null;
```

```
            FurnitureFactory furnitureFactory = null;

            RestaurantType restaurantType = RestaurantType.Chinese;

            switch (restaurantType) {
               case Indian:
                 restaurantFactory = new IndianRestaurant();
                 break;
               case Italian:
                 restaurantFactory = new ItalianRestaurant();
                 break;
               case Chinese:
                 restaurantFactory = new ChineseRestaurant();
                 break;
            }
            menuFactory = restaurantFactory.createRestaurantMenu();
            menuFactory.getEntireMenu();

            furnitureFactory = restaurantFacto-
ry.createRestaurantFurniture();
            furnitureFactory.getListOfFurnitureVendors();
      }
}:
```

When this code is executed it displays Chinese restaurant menu list and Chinese furniture vendor list.because restaurantType is set to Chinese restaurant type (`RestaurantType restaurant-Type = RestaurantType. Chinese;`):

```
##########################
Chinese Menu

ChineseAppetizer1
ChineseAppetizer2

ChineseMainCourse1
ChineseMainCourse2

ChineseDesert1
ChineseDesert2

##########################
Chinese Furniture Vendors
  Vendor 1
  Vendor 2
  Vendor 3
```

This example demonstrate how much work should be done in case when existing Abstract Factory model should be extended by new product.

Note that it is not necessary to change anything inside of `MenuFactory` class and corresponding subclasses.

Conclusion

Design Patterns becomes synonym for good design and implementation of best practice, as well as common mechanism for recording and sharing design knowledge and experience. However, one should never forget that Design Patterns are templates. Each Design Pattern implementation should be adapted to the particular context. There is not available only single solution to a particular problem. Rather are offered different solutions which are dependent of problems complexity.

Even it can be argue that design patterns are increasing complexity, proper patterns implementation would improve code flexibility and reusability and increased complexity would be paid in the project maintenance phase, as well as during project development and defects correction or changed and modified requirements implementation.

There is also another well know issue in case when there are available number of different solution. If there are more solutions and solutions variants available then it will require longer time to choose solution. If there is more combination then it will take longer time to compose desired solution. It will take even more time to test each of solution and different combinations and variations and all these will cause overall project costs increasing.

"*Unfortunately, there is no single design strategy that is right for all situations. This is due to the competing needs in software design to eliminate excessive redundancy and excessive complexity.*" (Trowbridge et al., 2003, p. 51)

Current software development is heavily based on using tools and frameworks. Many patterns are hidden in the frameworks and developers are using patterns even many are not aware which kind of pattern it is. For example Java EJB 3.0 is implementing Dependency Injection pattern and new Java persistence implementation Java Persistent API (JPA) is using the DAO Design Pattern for separating object persistence and data access logic from any particular persistence mechanism or API.

Different vendor can implement the same pattern differently. For example Microsoft implementation of Model View Controller (MVC) pattern is different than Java SUN implementation. In case of Microsoft MVC pattern implementation the Controller is merged into a View and Observer pattern is used to notify View when Model is changed. This implementation is called the Document-View variant.

Learning how to use patterns and adapting templates to the particular context can be a challenge. Each problem can be solved by different patterns. If there are more solutions it can take longer to make decision. Choosing right one cannot be decided out of a context in which pattern shall be used. Problem complexity and granularity as well as observing what other people are doing are most important in making right decision.

As well as development is an iterative process, so it is using of patterns too. As Martin Fowler states (1999):

"*Once you've made it, your decision isn't completely cats in stone, but it is more tricky to change. So it's worth some upfront thought to decide which way to go.*" (p. 30).

Continues refactoring would help to improve code design and reusability.

Even Design patterns are recommended as best practices they are not guarantee to project success. Design Patterns are mostly related to the component internal design and interactions with other components. Creating successful project is more than art of programming and is focused to the satisfying customers' requirements and delivery of project on time and according to budget constraints.

Further Work

In this article are described mostly Creational Design Patterns. The Structural and Behavioral Design Patterns are next logical step to learn about first 23 software design patterns, that are often referenced as basic design patterns.

Martin Fowler in his book “Patterns of Enterprise Application Architecture” (2003) presented another 51 design patterns.

POSA (Buschmann et al., 1996) presented another software design pattern.

There are a lot of patterns to learn about and what is more important to analyze and understand when and how these could be applied to solve real world issues.

Understanding of the Anti-Patterns is equally important as well as code refactoring (Fowler, 1999). Brown et al. book, The "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis" (1995), can be a good start point to future reading and research.

Software requirements are subject of continues changes and to be satisfied requires application code changes. Each implementation change could require changes in the current pattern implementation. If changes complexity is high it could require different design pattern or evolving of existing design pattern to more complex implementation.

Also it can be very interesting to execute benchmark test on the code that is implemented by using design patterns and code that is optimized to perform fast and is implemented without using design patterns.

These implementations should implement the same solution so it can be possible to compare what are difference in the execution time, and using of other resources. This test can be extended by using different patterns and changing implementation complexity.

Collected results can be used to analyze costs that are related to design pattern. However, that kind of analyses cannot be reliable if there is not involved cost analyses related to the code maintenance, defect correction, implementation of change orders and refactoring, improving design of existing code.

Refactoring and patterns is also very interesting further work. This article has mentioned refactoring but did not use time to explain and illustrate this important part of software development by examples. This can be subject of further work.

References

Brooks, F. P. (1975). *The mythical man-month: Essays on software engineering*. Addison-Wesley.

Brown, W., Malveau, R., McCormick, H., & Mowbray, T. (1998). *AntiPatterns: Refactoring software, architectures, and projects in crisis*. J. S. Wiley and Sons, available at http://www.antipatterns.com/briefing/sld001.htm

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-oriented software architecture, Volume 1: A system of patterns*. J.S. Wiley and Sons

Buschmann, F., & Henney, K. (2008). Pattern-oriented software architecture. *The essence of modern software engineering,, OOP 2008, Software meets business*, Munich. Retrieved from http://www.sigs.de/download/oop_08/Buschmann%20Mo2%20Patterns_OOP.pdf

DeMarco, T. (1979). *Structured analyses and system specification*. Prentice Hall PTR.

Fowler, M. (1999). *Refactoring improving the design of existing code*. Addison-Wesley.

Fowler, M. (2003). *Patterns of enterprise application architecture*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns elements of reusable object-oriented software*. Boston: Addison-Wesley. Partially available at http://c2.com/cgi/wiki?DesignPatternsBook

Google Knoll. (2010). *Top 100 best software engineering books, ever: The full list*. Available at http://knol.google.com/k/top-100-best-software-engineering-books-ever#

HistoryOfPatterns. (2010). *History of patterns*. Available at http://c2.com/cgi-bin/wiki?HistoryOfPatterns

Jensen, K., & Wirth, N. (1974, 1985). *User manual and report*. Springer-Verlag.

Kernighan, B. W., & Ritchie, D. M. (1978, 1988). *The C programming language* (2nd ed.). Prentice Hall.

Knuth, D. E. (1968). *The art of computer programming*. Addison-Wesley.

Koenig, A. (1995). Patterns and antipatterns. Article collected and introduced by Rising, L. (1998), *The patterns handbook*. Cambridge University Press. Available at (http://books.google.com/books?id=HBAuixGMYWEC&pg=PT1&dq=0-521-64818-1&hl=en#v=onepage&q=0-521-64818-1&f=false)

Microsoft. (2002). *Application architecture for .NET: Designing applications and services*. Microsoft Corporation. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/distapp.asp

MultitierArchitectureWikipedia. (2010). *Multitier architecture*. Available at http://en.wikipedia.org/wiki/Multitier_architecture

Oracle SUN Developer Network. (2010). *Core J2EE patterns: Patterns index page*. Available at http://java.sun.com/blueprints/corej2eepatterns/Patterns/

Srinivasan, R. (2010). *Undocumented 'lava flow' antipatterns complicate process*. Available at http://www.icmgworld.com/corp/news/Articles/RS/jan_0202.asp

Stroustrup, B. (1991). *The C++ programming language*. Addison-Wesley.

Trowbridge, D., Mancini, D., Quick, D., Hohpe, G., Newkirk, J., & Lavigne, D. (2003). *Enterprise solution patterns using Microsoft .NET 2.0*. Available at http://msdn.microsoft.com/en-us/library/ff647095.aspx

Wikipedia-Anti-pattern. (2010). *Anti-pattern*. Available at http://en.wikipedia.org/wiki/Anti-pattern#cite_note-2

Biography

Aleksandar Bulajic received a Master's of Science in IT (With Distinction) degree from Liverpool University, England. He received a Bachelor (BA) degree from Sarajevo University.

He is currently PhD student at Metropolitan University Belgrade, Faculty of Information Technology. He has Danish citizenship and is working for IBM Denmark.

Aleksandar Bulajic has more than 27 years of working experience in the IT industry. He has executed many different roles during an IT project's lifetime. He has worked as a Developer, Tester, Test Manager, Project Manager, IT Architect, Requirement Manager, IT Designer, Customer Supporter, Development Supporter, and is able to execute many other roles. He has developed projects under UNIX, Windows, and mainframe platforms.

Aleksandar Bulajic has collected a large amount of experience from the real projects, implemented for known customers, and deployed in production. His technical competence is very wide. He is able to help customers specify business requirements and help developers to understand project requirements, and establish development and test environments, get started with new technologies, design databases and create automated testing tools, register and track software issues, establish necessary infrastructure, and analyse application bottlenecks, and solve application development and production issues.

His strength is in his persistence to achieve project goals despite all kind of issues that arise during a project's lifetime.

His primary interest is technology and technology implementation. He is focused on automating development and maintenance tasks, and creating standardized frameworks, tools, and processes and methods, that can reduce overall project expenses and increase the capabilities and effectiveness of each particular developer and project team member, in the development and maintenance phases, and also ensure that the final product satisfies quality standards as much as it is possible in each particular case.

His technical papers and articles were published and presented at International IT conferences and at IBM Intranet:

1. The 8th World Multi conference on Systematic, Cybernetics and Informatics (SCI 2004), Orlando, Florida, USA, July 18-21, 2004.,
2. InSITE, Information Technology Education Joint Conference, Rock Hampton, Old Australia, June 2004
3. IBM Intranet, "Excel 2003 and Visual Basic 2008 Tips", available at IBM Intranet https://w3-03.sso.ibm.com/services/practitionerportal/ppServlets/displayDocument.wss?syntheticKey=J813576Y12421W52, 15-07-2009
4. InSITE, Information Technology Education Joint Conference, Novi Sad, Serbien, June 18-23, 2011

Contact info: Aleksandar Bulajic, Buddingevej 48, 2800 Kongens Lyngby, Denmark, e-mail: LANB@45.dk