Towards A Pattern Language for Developing Stable Software Patterns- Part I

Haitham Hamza[1] and Mohamed Fayad[2]

[1]*Computer Science and Engineering Dept., University of Nebraska-Lincoln*
Lincoln, NE 68588, USA
hhamza@cse.unl.edu
[2]*Computer Engineering Dept., College of Engineering, San José State University*
One Washington Square, San José, CA 95192-0180
m.fayad@sjsu.edu

ABSTRACT

In previous work we have discussed some of the problems that face the contemporary analysis patterns, and proposed the new concept of "Stable Analysis Patterns" as a solution to these problems [2,3,5]. Since that time we have applied the same concepts to design as well as analysis, resulting in the broad concept of Stable Software Patterns, with specialized concepts for Stable Analysis Patterns and Stable Design Patterns. Stable software patterns is a new approach for developing patterns based on software stability concepts [6]. Our long term goal is to develop a pattern language that deals with all the aspects related to the concept of stable patterns (such as stable patterns construction, documentation, and applications). This paper presents our second step towards this goal by presenting part of the ultimate pattern language. In this paper we first provide an overview of the concept of stable patterns. Second, we provide the big picture of the overall pattern language. Finally, we present five patterns as part of the pattern language.

1. INTRODUCTION

In a previous work, we have discussed different problems in contemporary analysis patterns, and as a solution to these problems, we have proposed the concept of *Stable Analysis Patterns* [2,3,5]. Stable analysis patterns are kinds of patterns that are built using the software stability concepts introduced in [6,7]. The main objective of stable analysis patterns is to provide a model that captures the core aspects of the problem. Capturing the core of the problem is mandatory to designing the right solution. A brief overview of stable analysis patterns is given in the next section.

Later on, we have generalized the concept of stable analysis patterns to accommodate design patterns as well (*Stable Design Patterns)*. This generalization has led to the new broad concept of *Stable Software Patterns*. For the purpose of this paper, we do not need to differentiate between analysis and design; thus, we just use the general term stable patterns.

Based on the number of EBTs, stable patterns can be categorized into two main groups: *architectural stable patterns*, and *stable atomic knowledge (SAK) patterns.* An architectural pattern consists of more than one EBT whereas SAK pattern consists of only one EBT. Figure 1 illustrates the different categorizes of stable patterns.

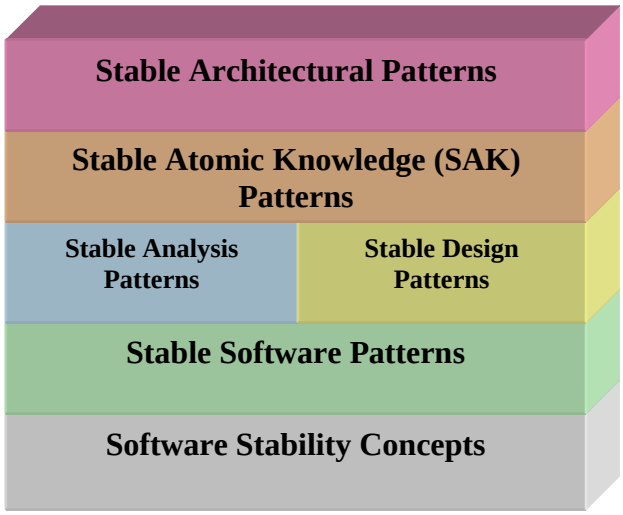

Figure 1. Stable software patterns concepts and categorizes

The set of patterns presented in [4] presents our first attempt towards the development of the pattern language for stable patterns. Some of the patterns presented in [4] can be applied in the broader context presented in this paper.

2. SOFTWARE STABILITY AND STABLE PATTERNS: AN OVERVIEW

Software stability concept [6] is a layered approach for developing software systems. In this approach, the classes of the system are classified into three layers: the Enduring Business Themes (EBTs) layer [1], the Business Objects (BOs) layer, and the Industrial Objects (IOs) layer. Figure 1 depicts the three layers of software stability approach.

Based on its nature, each class in the system is classified into one of these three layers. EBTs are the classes that present the enduring and core concepts of the underlying industry or business. BOs are the classes that map the EBTs of the system into more concrete objects. BOs are semi-conceptual and externally stable, but they are internally adaptable. IOs are the classes that map the BOs of the system into physical objects. For instance, the BO "Agreement" can be mapped in real life as a physical "Contract", which

is an IO. The detailed properties that characterize EBTs, BOs, and IOs can be found in [7].

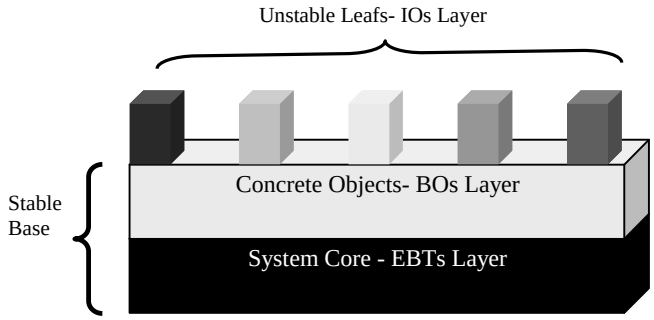

Figure 2. Software stability approach layers

Stable analysis pattern introduced in [2,3,4], is a new approach for developing patterns by utilizing software stability concepts. Stable analysis pattern was proposed as a solution for the limitations of contemporary analysis patterns we have discussed in [2]. The goal of stable analysis pattern was to develop models that capture the core knowledge of the problem and presented it in terms of the EBTs and the BOs of that problem. Consequently, the resultant pattern will inherent the stability features and hence can be reused to capture the essence the same problem whenever it appears.

3. THE PATTERNS LANGUAGE: A BIG PICTURE

The objective of the overall pattern language is to cover all the essential aspects related to concept of stable patterns. The process of developing stable patterns involves four main steps: *Developing Stable Patterns, Documenting Stable Patterns, Testing and Validating Stable Patterns*, and finally, *Applying Stable Patterns.* For each of these four steps there are different sets of patterns that interact together to accomplish the goal of this step.

Figure 3 depicts the overall pattern language structure. In the figure, the main four steps are presented in orange boxes. The blue boxes present the major issues that are needed to accomplish the goal of the steps they belong to. For instance, the first step Developing Stable Patterns contains two main issues: Pattern Construction and Team Dynamics. Each issue is then accomplished and described using a set of patterns. For instance, under Pattern Construction, we have four patterns, green boxes. Each of these four patterns presents a step in the process of constructing the stable pattern.

It is important to note that the pattern language will consist of the green boxes only, as the other boxes are just used to classify patterns based on their objective. Green boxes with the red circles in the Figure 3 represent the part of the pattern language we address

in this paper. For simplification, we do not include all the detailed patterns and issues that are involved in each of the development process steps.

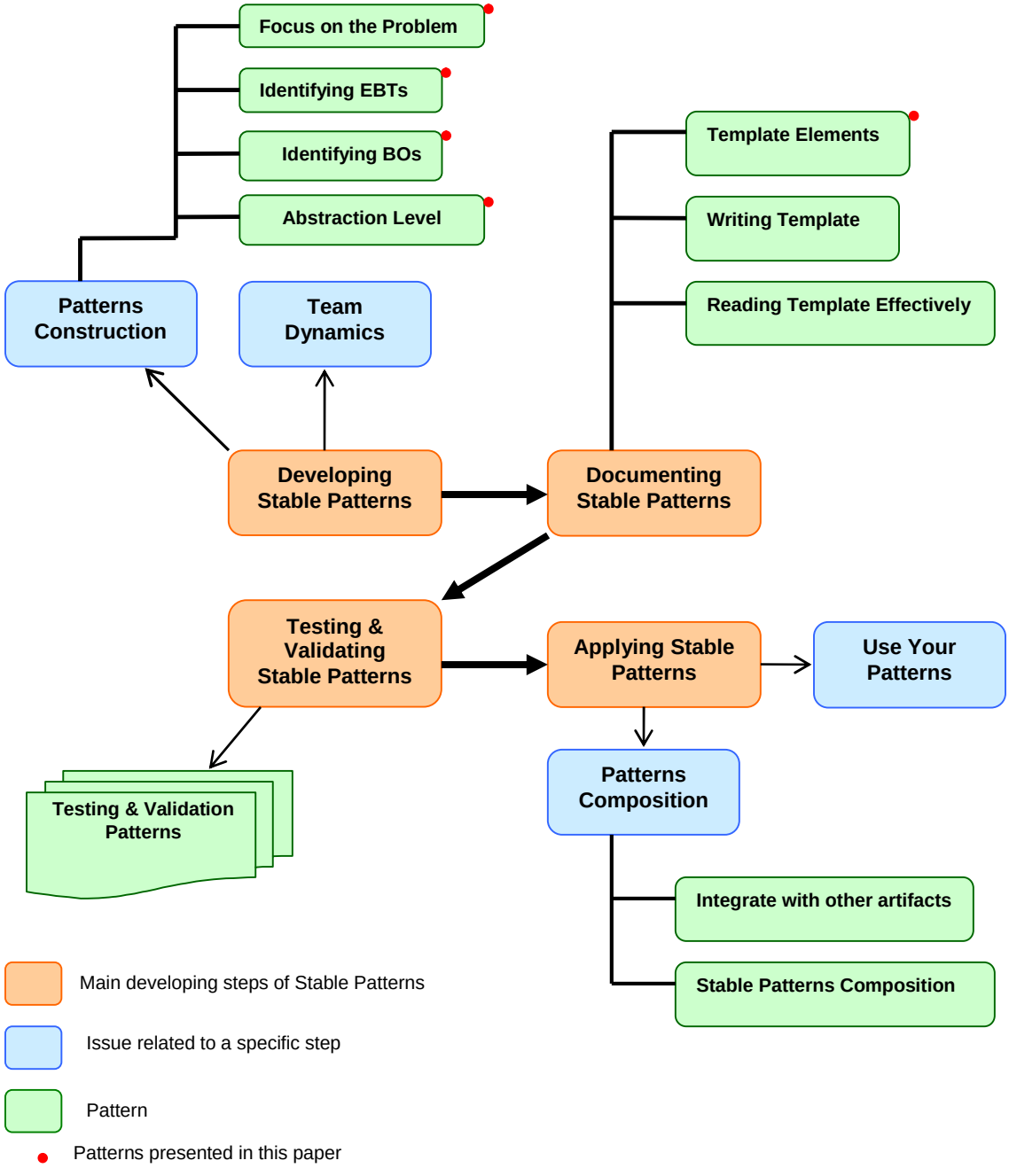

Figure 3. Description of the overall pattern Language *(Does not show all the patterns)*

4. PATTERNS LANGUAGE DESCRIPTION: PART I

In this section we describe five patterns from the overall pattern language. These patterns are marked in red circle in Figure 3. Table 1 below summarize the five patterns we present in this paper and gives the page number of the solution that each pattern presents for a quick reference.

Table 1. Summary of the Pattern Family

Pattern	Problem	Solution
Focusing on the Problem	How to focus on a specific problem that the analysis pattern will model?	Page 4
Identifying Enduring Business Themes	How to identify the enduring business themes of the problem?	Page 6
Identifying Business Objects	How to identify the business objects of the problem?	Page 7
Expressing Abstraction Levels	How to assemble the problem model components to build the stable pattern? How to define the relations between the identified EBTs and BOs?	Page 9
Documenting Stable Patterns	How to document stable patterns efficiently?	Page 10

PATTERN 1 – FOCUSING ON THE PROBLEM

Context

The reusability of a stable pattern is related to the number of problems it addresses. For instance if a pattern is used to model many problems, the generality of the resulting pattern will be reduced, since the probability of the occurrence of all the problems together is less than the probability of the occurrence of each problem individually. It is not always easy and safe to extract portion of the pattern to use in another problem. Focusing on a specific problem is one of the key factors that helps improve the reusability of the pattern.

The developed pattern is not intended to represent a model for a complete system; rather it models a specific problem that commonly appears within larger systems (these systems could belong to same or different domains). Systems, by their nature, combine many problems. Thus, they can be developed using a collection of patterns. Without decomposing a system into components, the pattern size becomes unreasonably complex, and the generality of the patterns is adversely affected. If a pattern is used to model an overly broad portion of a system, the generality of resulting patterns is sacrificed - the maxim holds: the probability of the occurrence of all the problems together is less than the probability of the occurrence of each problem individually. For example, modeling the "payment" problem with "buying a car" is not effective since the "payment" problem may appear in unlimited number of problems.

Problem

How to focus on a specific problem that the pattern will address?

Forces

- Certain groups of problems often appear together. As a result, they will be modeled as one problem. The resultant model may or may not be correctly modified to model these problems when they appear separately.
- In practice, not all of the small problems that we can separate are qualified to form practical stand-alone problems. There is a tradeoff between dividing the problem and the complexity of integrating smaller problems to model a larger problem.

Solution

Before we start modeling the problem we need to check whether or not this problem can be further divided into smaller, practical problems. The following questions help us to do so: “What is the problem that we need to solve?” “Can we divide this problem further into a list of smaller problems?” “Are there any known possible scenarios where these smaller problems can appear?”

If we can find practical scenarios for each of the smaller problems that we have separated, then we need to model each of them separately. If the smaller problems have no practical use, they should be grouped together.

Example

We consider the "account" problem to illustrate the idea of problem separation. It was not so long ago when the word "account" was merely used to indicate banking and financial accounts. Today, the word "account" alone becomes a vague concept if it is not allied with a word related to a certain context. For instance, besides all of the traditional well-known business and banking accounts, today we have e-mail accounts, on-line shopping accounts, on-line learning accounts, subscription accounts, and many others.

One model for the account problem is the *Account* pattern shown in Figure 4 [9]. This pattern models two different problems at the same time. The first problem is the "account" problem and the second problem is the "entry" problem. These are two independent problems. Even though they appear together in many applications, there is a possibility of having entries without an account or accounts without entries. Figure 5 and Figure 6 show some examples of accounts without entries and entries without accounts, respectively. We have developed a separate model for each of these problems. The developed models for both the Account and the entry can be found in [3].

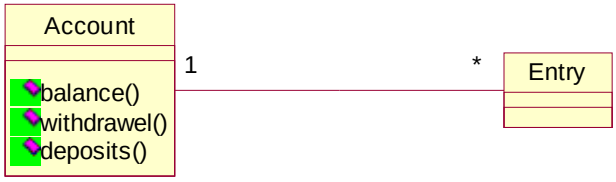

Figure. 4 *Account* pattern provided by Fowler [9]

(1) Free on-line services account: There are many on-line companies that provide free goods or services. For example, some companies provide learning software packages or instructional documents. In order to access these materials, these providers require you to create an account with the company. This account is simply a passport provided to enable you to access their service; you do not have anything in this account that can be considered to be your property. In fact, the only things that you can do with this account are the limited functions prescribed by the company that issued the account.

(2) Access account to the copy machine: Suppose that you have an account to access the copy machine in your school or work. This account is no more than a passport for you to use the copier. There are no entries in this case. (Note that in this example it is possible to use Fowler's pattern by changing the names of the behaviors in his patterns).

Figure. 5 Accounts without entries

The following table contains information about class schedules, at the University of Nebraska-Lincoln, Spring 2002. In this table, each piece of information forms an entry to the table. Here we do not need accounts in which to keep these entries.

Call #	Course Title	Cr Hrs	Time	Day	Room
2850	Computer Architecture	003	0230-0320p	M W F	Freg 112
2855	Software Engineering	003	0930-1045p	T R	Freg 111

Figure. 6 Entries without accounts

PATTERN 2 – IDENTIFYING ENDURING BUSINESS THEMES

Context
When developing a stable pattern, first we need to identify the core elements of the problem. These core elements present the enduring themes of the problem.

Problem
How to identify the Enduring Business Themes (EBTs) of the problem?

Forces
- EBTs should capture the core aspects of the problem; however, some EBTs capture the core knowledge of the problem within a specific context. Such EBTs should be discarded from the model.
- Being expert in the domain does not always guarantee an accurate generation for the relevant EBTs. For instance, a professional chef might mistakenly identify pots, pans, and refrigerator as an enduring business theme for modeling the kitchen while they are in fact Industrial Objects (IOs) [7]. Thus, experience is essential but not sufficient condition for extracting the correct EBTs in the problem we analyze.
- Even though many of the selected EBTs might appear strongly related to the problem at first glance, many of them in fact have nothing to do with the problem being modeled.
- Some of the EBTs might lack one or more of the EBTs essential properties [7]. In this case, we should re-identify them as BOs or IOs.

Solution
The following steps help in extracting the appropriate EBTs of the problem:

Step 1 Create Initial EBTs List
To create the initial list of the EBTs of the problem, answer the question: "What is the "problem" for?" In other words: "What are the reasons for the existence of the "problem"?"
The output of this step is the list of the initial EBTs of the problem. These EBTs are still tentative and some of them are not as strongly related to the problem as they might appear.

Step 2 Filter the EBTs List
Eliminate the redundant and irrelevant EBTs from the initial list. People usually unintentionally construct the initial EBTs list with a specific context in mind. The output of this step is a modified EBTs list, which is usually smaller than the initial list.

Step 3 Check the Main EBTs Properties
Examine the EBTs obtained in previous steps against the main essential properties of the EBTs. The typical procedure is to answer the following questions for each EBT in the list. The desired answer is written in **bold** beside each question:

- Can we replace this EBT with another one? **No.**
- Is this EBT stable internally and externally? In other words, does this EBT reflect the core aspects of the problem we are trying to model? **Yes.**
- Can we directly represent this EBT physically? **No.**

It is important to note that the EBTs should not have direct physical representations (IO); otherwise they should be considered BOs instead. (Refer to the software stability model architecture shown in Figure 2). For example: "Agreement" is a concept and one can see it as an EBT. However, "Agreement" also has a direct physical representation (for instance "Contract"). Therefore, "Agreement" is not an EBT, it is a BO. Any EBT that does not satisfy one of these properties should be eliminated from the list.

PATTERN 3 – IDENTIFYING BUSINESS OBJECTS

Context
When developing a stable analysis pattern, and after identifying the EBTs of the problem, we need to identify the Business Objects (BOs) of the problem.

Problem
How to identify the business objects of the problem?

Forces
- In some cases, it is not obvious whether the object is an EBT or BO. For instance, "Agreement" can be considered an EBT since it presents a concept. However, it is a BO.
- After the EBTs of the problem have been identified, the conceptualization becomes more involved since the BOs of the problem must be based on the defined EBTs. This makes it difficult to extract the BOs.
- There is rarely a one to one mapping between the EBTs of the problem and its BOs. It is possible for EBTs to have no direct mapping to the BOs and for the BOs to have no direct mapping to the EBTs. Often one EBT can be mapped into several BOs.
- In addition to the main BOs that we can identify for the problem, it is possible to have some hidden BOs that have no direct relationship with the defined EBTs. Instead, they are related to the main BOs and to the other hidden BOs in the problem.

Solution
One approach that helps extract the appropriate BOs of the problem is to follow the following four steps:

Step 1 Identify the main BOs of the problem
In this step we identify the main set of BOs that are directly related to each of the EBTs we have in the problem. There could be one or more BOs corresponding to each EBT in the problem. However, some of the EBTs may have no corresponding BOs.

The main set of BOs of the problem can by identified by answering one or more of the following questions for each EBT: [Note: some questions do not apply for some of the EBTs. This depends on the nature of each EBT]

- How can we approach the goal that this EBT presents?
  [For example: To achieve the goal of the EBT Organization, we can use, the BO Schedule. Another example: for the EBT Negotiation we need the BOs AnyContext, and AnyMedia to perform the negotiation].

- What are the results of doing/using this EBT?
  [For example: for the EBT Negotiation, the eventual result is to reach an Agreement so this is one possible BO that maps this EBT].
- Who should do/use this EBT?
  [For example: The BO Party does/ uses Negotiation. This Party can be a person, a company, or an organization. Therefore, Party is one possible BO that maps the EBT Negotiation].

Step 2 Filter the main BOs List

Purify the main BOs identified in the previous step. The objective of this step is to eliminate the redundant and irrelevant BOs from the initial list. One way to achieve this goal is to debate the listed BOs with a group.

Step 3 Identify the hidden BOs of the problem

Identify the hidden BOs of the problem. These BOs are named "hidden" because they have no direct relationships with any of the EBTs of the problem. Thus, we cannot extract them in the first two steps we have performed.

For example, suppose we need to model a simple transportation system that offers transportation services for different types of materials (for example, gas, water, etc.). One possible EBT is Transportation. One possible BO that maps this EBT is Transport. A possible IO that can physically represent this BO is Trucks. In this problem, one possible hidden BO is Materials. We do not have a direct EBT that the BO Materials can be mapped to; however, there is a clear relationship between the two BOs Transport and Materials.

Before thinking about the hidden BOs in the problem, visualize a provisional scenario for each EBT and its corresponding BOs. Then answer the question "What is still missing in the problem?" Usually the answer to this question is the list of the hidden BOs of the problem. Some problems do not have any hidden BOs, especially in the case of the small-scale problems.

Step 4 Check the characteristics of the BOs

This step is to make sure that the identified BOs satisfy the main BOs characteristics. BOs are:

- Partially tangible.
- Externally stable and should remain stable throughout the life of the problem.
- Adaptable (thus, they might change internally).
- Physically represented by IOs.

Pattern 4 – Expressing Abstraction Levels

Context
Stable patterns can be generally classified into two main categories: simple and composite patterns. A simple analysis pattern is a pattern that just consists of classes and no sub-patterns exist. On the other hand, composite stable analysis pattern consists of both classes and sub-patterns. Simple patterns are said to have one level of abstraction (such as the conventional class diagram), while composite analysis patterns might have several levels of abstraction depending on the structure of its sub-patterns.

Problem
How to express the abstraction levels of the composite stable pattern?

Forces
- Stable patterns can be either simple or composite. Differentiating between both kinds is essential to differentiate between both kinds when they are used in the development. Current modeling tool and techniques does not support stable patterns and, hence, we have to deal with this challenge to express our new concept clearly using the existing modeling tools.
- There is a tradeoff between the complexity and the clarity of the stable pattern. That is, some composite patterns may involve several sub-patterns and hence expressing the detailed structure of each sub-pattern in the first abstraction level yields a large pattern structure that complicates the understanding of that pattern.

Solution
In stable patterns, we differentiate between two main participants in the pattern model, classes and patterns. Classes are defined as in any traditional Object-Oriented class diagram. On the other hand, patterns present a second level of abstraction in the model, where each pattern is by itself another model that contains classes and, in some cases, other patterns (our practical experience shows that going beyond two abstraction levels would unnecessarily complicate the pattern, and might introduce some useless classes to the system).

A class in a stable pattern could be one of the five following kinds: an EBT, a BO, an IO, a sub-pattern and EBT, or a sub-pattern and BO. Therefore, each class in the stable pattern should have one of the following tags: EBTs, BOs, IOs, Pattern-EBT, or Pattern-BO. Note that there is no need for a tag: Pattern-IO, because the nature of IO as unstable and replaceable artifacts prevents them from being a pattern that can be reused over and over as in the case of EBTs and BOs.

Example
The *Negotiation* pattern shown in appendix A is an example of the first abstraction level of a composite stable analysis patterns. The *Negotiation* pattern has one class named

Negotiation and it is an EBT, and four sub-patterns: *AnyMedia*, *AnyAgreement*, *AnyContext*, and *AnyParty*.

Each of these sub-patterns has its own separate structure and hence can be used in other application independent of the *Negotiation* pattern itself. The *AnyMedia* sub-pattern, for instance, presents another stable pattern and, hence, it forms a second abstraction level in the *Negotiation* pattern. The structure of the *AnyMedia* sub-pattern is given in Appendix B. The *Negotiation* pattern is a composite analysis pattern, while *AnyMedia* is a simple analysis pattern.

PATTERN 5 – DOCUMENTING STABLE PATTERNS

Context

One crucial factor in utilizing patterns in general is the proper documentation of the pattern. Poor documentation will result in inefficient utilization of the pattern and perhaps a complete misunderstanding of the pattern. . Several design pattern styles have been used to document design patterns [10], [11]. On the other hand, analysis patterns do usually use the traditional style, and sometimes a mixture of these design patterns documentation styles. However, the difference in the nature between stable analysis and design patterns and their traditional counterparts makes the documentation of stable patterns an issue that that should be considered.

Problem

How to document stable patterns efficiently?

Forces

- Analysis patterns are conceptual models that are difficult to understand by their nature. In addition, design and analysis are very different and techniques to understand one of them do not necessarily help in understanding the other. Therefore, great care needs to be taken when developing a template to document stable design pattern and stble analysis patterns.
- In stable patterns, there are no Industrial Objects (IOs) presented. IOs usually simplify the understanding of pattern applicability. However, attaching IOs to the pattern will limit its applicability to specific domain or application. For instance, in the *Negotiation* pattern (Appendix A), the role of the sub-pattern *AnyMedia* might not be obvious from just reading the first level of abstraction (i.e. the *Negotiation* pattern itself); however, if we have replaced this sub-pattern by a simple media class such as an email or a phone, the role becomes more obvious. But, doing so will limit the pattern applicability to these applications or domains that usually negotiate through the email and/or the phone. The abstraction of the negotiation media is essential to broaden the scope of the pattern. The tradeoff between simplicity and generality should be considered in stable analysis patterns.
- Presenting Sub-patterns as single class in the first abstraction level of the pattern might make them hard to understand and document. For instance, in the *Negotiation* pattern, it might not be easy from the first glance to understand what the role of the sub-pattern *AnyMedia* is. Also, when we document the *Negotiation* pattern, it is required that we express the role of the sub-pattern into one single abstract role leaving the detailed role of the components of each class in the sub-pattern to the second abstraction level of the pattern. Coming up with this single abstract role is not always straightforward and need to be done carefully; otherwise, the pattern will not be clear.

Solution

We propose some addition to the conventional design pattern documentation templates. Few adjustments have been done to the following fields: Context, Problem, Forces, Applicability, Known uses, and Consequences.

The Solution field needed to be modified to capture the different abstraction levels presented in composite stable patterns. In our template, the Solution consists of three main parts:

a) **Static structure**. Gives the class diagram of the pattern. It also introduces briefly each class and its role. Association classes, constraints, interfaces, tagged values, and notes must be included in the class diagram. Static structure reveals the structure of the pattern that contains it participants and the relationships among them. In this paper we use UML notation and guidelines for generating the pattern structure. However, it is possible to apply any other OO modeling languages (Booch, Odell, etc) the developer wish to use in developing stable patterns.

b) **Participants.** The participants of any stable pattern can be classified into two main categories: Classes and Patterns.

c) **CRC Cards**. Summarizes the responsibility and collaboration of each participant. The CRC names the class, responsibility, and its collaborations. The CRC card also names a role for each class, which is useful for identifying the class responsibility. Each class should have only one responsibility, and that responsibility should be unique. The collaboration consists of two parts: clients and server. Clients section contains all the classes that collaborate and have relationships with the named class. The Server section lists all the services that the named class can provide to its clients [8]. It is worth to point out that in documenting CRC – cards for stable patterns we deal with any patterns that are included within the main pattern itself as a class. That is, each sub-pattern will be represented by a CRC-card that documents its responsibility and collaborations as a black box. To avoid any confusion, and for simplicity, we do not care about how the sub-pattern handles its responsibility according to its internal structure, all what we care about here is that this sub-pattern will perform the task as a block, leaving the other details to the second abstraction level of the pattern description. For instance, the CRC-cards of the sub-pattern *AnyMedia* will show the details of each class in the black box *AnyMedia*.

Class/Pattern Name (Class/Pattern Role)		
Responsibility	**Collaboration**	
	Clients	**Server**
A single responsibility for this Class/Pattern should be listed here briefly.	A list of all the Classes/Patterns that have a relationship with the current Class/Pattern.	List of all the servers that this Class/Pattern provides.

The following is an example of a CRC-Card for the pattern *AnyMedia* exists in the context of negotiation.

Pattern **:AnyMedia** (Connector)-Pattern		
Responsibility	**Collaboration**	
Communicates negotiation issues between negotiators.	**Clients**	**Server**
	Negotiation	connectParties() display() illustrate()

As we have mentioned in the Forces listed earlier, it might be hard to understand the first abstraction level of the stable pattern (if the pattern has more than one abstraction level). Because of the tradeoff between pattern simplicity and generality, we have developed some techniques to make as easier to understand the pattern. Conventional applicably field that is used in traditional pattern templates might help but is not sufficient. Therefore, we have introduced what we called *Role-based Instance Diagram and Role-based Scenario.* For each applicability example that we state for a stable pattern these two new artifacts were found to provide good visualization for the proposed pattern.

A *Role-based Instance Diagram* is a diagram that shows how each sub-pattern/class in the main pattern diagram can be instantiated in the examples of the pattern applicability. It also shows how the pattern components interact with each others in the application. On the other hand, a *Role-based Scenario* is a textual description of the role-based instance diagram. Each component in the pattern plays its role in the scenario. We find this helps reader to gain better understanding of the problem that the pattern addresses. In the scenario, we use the exact names of the classes, sub-pattern, and roles specified on a relationship between them in the stable object model in the solution section.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a collection of patterns that forms the first step towards a pattern language for developing stable patterns. We presented five patterns: *Focusing on the Problem, Identifying Enduring Business Themes, Identifying Business Objects, Expressing Abstraction Levels*, and *Documenting Stable Patterns.* In the future, we will present another part of the pattern language that address other issues related to the development of stable patterns.

ACKNOWLEDGEMENTS

We thank Marshall Cline, our shepherd, for his valuable comments and suggestions that have improved this paper.

We would like to thank Bruce Whitenack, the shepherd of our PLoP 2002 pattern language, for his useful comments and for his valuable suggestions.

Thanks to the members of the PLoP '02 Workshop, Sweet Home Allerton, for taking time to review the proposed patterns: Ahmed Mahdy, Linda Rising, Danny Dig, Joel Jones, Shasha Wu, Toby Sarver, Masao Tomo, Joe Yoder, and. Martine Devo. Also we would like to thank Marshall Cline for reviewing the first draft of the PLoP 2002 pattern language.

REFERENCES

[1] Cline, M., Girou, M.: Enduring Business Themes. Communications of the ACM, Vol. 43, No. 5, May 2000, pp. 101-106

[2] H. Hamza "A Foundation For Building Stable Analysis Patterns." Master thesis. University of Nebraska-Lincoln, 2002

[3] H. Hamza. "Building Stable Analysis Patterns Using Software Stability". 4th European GCSE Young Researchers Workshop 2002 (GCSE/NoDE YRW 2002), October 2002, Erfurt, Germany.

[4] H. Hamza and M.E. Fayad. "A Pattern Language for Building Stable Analysis Patterns", *9th Conference on Pattern Language of Programs (PLoP 02)*, Illinois, USA, September 2002.

[5] H. Hamza and M.E. Fayad. "Model-based Software Reuse Using Stable Analysis Patterns" ECOOP 2002, Workshop on Model-based Software Reuse, June 2002, Malaga, Spain.

[6] M.E. Fayad, and A. Altman. "Introduction to Software Stability." Communications of the ACM, Vo. 44, No. 9, September 2001, pp 95-98.

[7] M.E Fayad. "Accomplishing Software Stability." Communications of the ACM, Vo. 45, No. 1, January 2002, pp 95-98.

[8] M.E. Fayad, V. Stanton, and H. Hamza. "A New Look At the CRC Cards." http://www.activeframeworks.com

[9] M. Fowler, "Analysis Patterns: Reusable Object Models", Addison-Wesley, 1997.

[10] E. Gamma et al. *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.

[11] Buschmann, F. et al., *"Pattern-Oriented Software Architecture, A System of Patterns"*, John Wiley & Sons Ltd, Chichester, 1996.

APPENDIX A: NEGOTIATION PATTERN: BRIEF DOCUMENTATION

Context

Negotiation is a general concept that has many applications. In our every day life, there are various situations where negotiation usually takes place. For instance, buying or selling properties usually involves some sort of negotiation (e.g. buying or selling a home or a car). In software systems, negotiation also appears frequently in the development of different applications. Developing software for online auctions and shopping might involve the negotiation of the price and/or the negotiation of different product aspects.

More technically, negotiation becomes an essential part in the development of next generation Web-based devices and appliances. Today, devices that need to access the Web diverge greatly in their capabilities so negotiation algorithms between client agent and servers play a fundamental role in helping servers decide which representation of a document a device should be given. Therefore, having a stable pattern that can model the basic aspects of a negotiation problem would make it easier for the developer to build their system by reusing and extending this pattern.

Problem

The complexity of modeling generic concept such as negotiation arises from the fact that these concept while have the same implication whenever they appear, each application has its own specific, and sometimes unique, requirements depending on the application nature and objective. The context section before and the forces section next illustrate how can the requirements of the negotiation process differs from one application to another.

The fact that negotiation concept does span a wide range of spectrum of heterogeneous applications, along with the fact that the negotiation concept itself does not change whenever it appears, both makes the development of a model that captures the core knowledge of the negotiation concept both desired and challenging. Developing such generic and accurate model is not easy and this leads to the main question: How can we build a negotiation model that can be used to model the negotiation problem in any application?

Solution

The proposed solution is to focus on the concept of negotiation trying to extract the main components of the negotiation concept, leaving other domain-specific and/or application-specific components away form this core model. The basic components are represented in generic way that allows the developer to utilize them according to the needs of his/her applications.

Figure A.1 below shows the object diagram of the *Negotiation* pattern. The *Negotiation* pattern consists of the following participants:

Classes:

- *Negotiation:* Represents the negotiation process itself. This class contains the behaviors and attributes that regulate the actual negotiation process.

Patterns:

- *AnyAgreement:* Represents the result of the negotiation. The ultimate goal of any negotiation is to reach an agreement. Thus, this object presents a core element in any negotiation. It is important to note that in many cases negotiation ends with no agreement and thus it is considered to be failed (the seller of the car did not agree on the price proposed by the buyer and vise versa), however, in this case we expect that the agreement should provide this result by whatever mechanism. So one can view the agreement object as the result of the negotiation, which is not necessary a successful result.
- *AnyParty:* Represents the negotiation handlers. It models all the parties that are involved in the negotiation process. Party can be a person, organization, or a group with specific orientation.
- *AnyMedia:* Represents the media through which the negotiation will take place. For instance, one can negotiate the price of a good over the phone. Others might use an email or a mail to negotiate specific issues in their business.
- *AnyContext:* Represents the matters to be negotiated. If we are buying a home, many issues could be negotiated. For instance, the price of the home, the payment procedure, etc. Defining the issue to be negotiated is an essential element of any negotiation process; otherwise, negotiation will have no meaning.

The prefix 'any' that we used herein indicates that this is another pattern that provides an abstract model for the notion it precedes. For instance, AnyParty is a stand-alone stable pattern that models the party notation and, hence, can be used to model any party in any applications. The detailed structure of this pattern is out of the scope of this paper

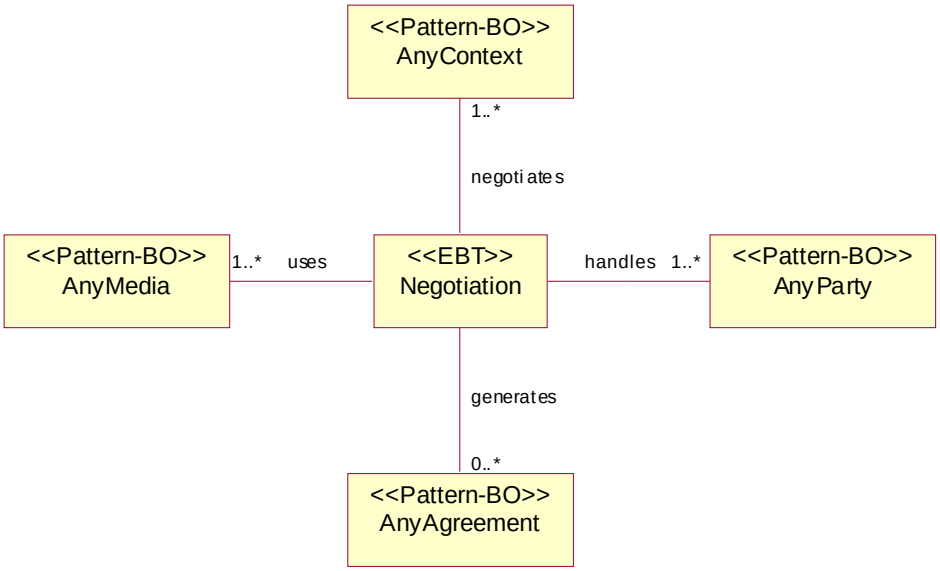

Figure. A.1 *Negotiation* pattern stable object model

APPENDIX B: ANYMEIDA PATTERN: BRIEF DOCUMENTATION

Problem

How to build a model that can be used to present any media in any application?

Context

The pattern can be used to model any media of any type and any kind. For instance, a media type such as multimedia can be of any kinds (e.g., image, voice, etc.).

Solution and Participants

Figure B.1 shows the object diagram of the *AnyMedia* pattern. The shown model gives the high abstract level of view for the proposed model.

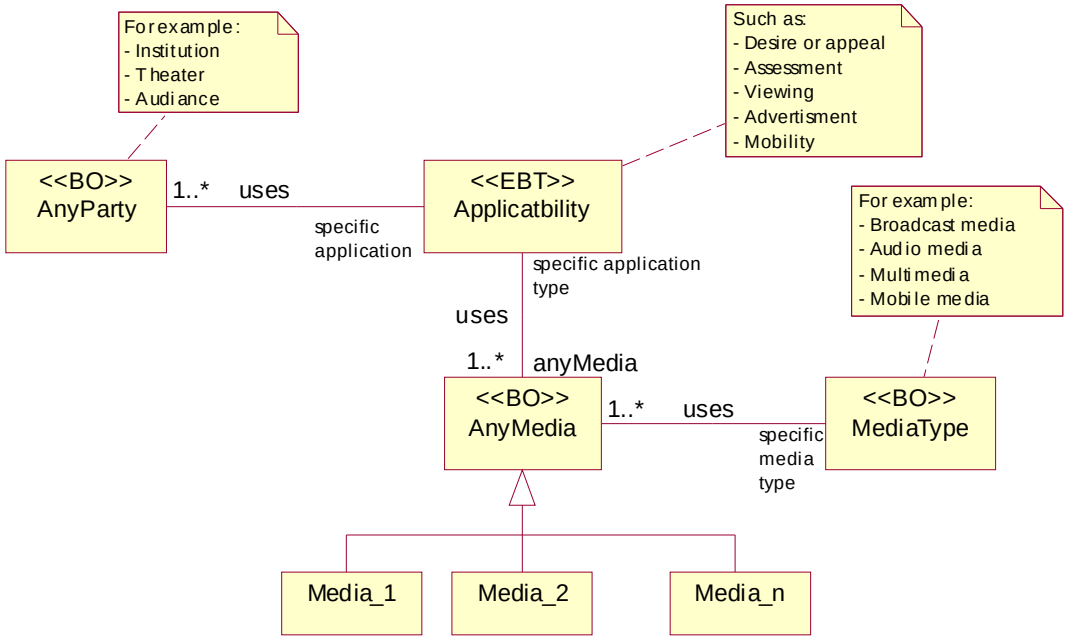

Figure B.2. *AnyMedia* pattern object diagram

Participants

The participants of the *AnyMedia* pattern are:

Classes:

- *AnyMedia.* Identifies the media to be used
- *MediaType.* Specifies the type of the used media.
- *ApplicationType.* Describes the purpose of which the media is used.
- *AnyParty.* Represents user of a specific media