2006

Building systems using patterns : creating knowledge maps

Huascar A. Sanchez
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Sanchez, Huascar A., "Building systems using patterns : creating knowledge maps" (2006). *Master's Theses*. 2976.

DOI: <https://doi.org/10.31979/etd.ut43-ysfv>

https://scholarworks.sjsu.edu/etd_theses/2976

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

BUILDING SYSTEMS USING PATTERNS: CREATING KNOWLEDGE MAPS

A Thesis

Presented to

The Department of Computer Engineering

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Interdisciplinary Master of Science

By

Huascar A. Sanchez

August 2006

UMI Number: 1438591

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 1438591

Copyright 2007 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© 2006

Huáscar A. Sánchez

ALL RIGHTS RESERVED

APPROVED BY THE COMPUTER ENGINEERING DEPARTMENT

Dr. M.E. Fayad, Professor at the Computer Engineering Dept.

Dan Harkey, Professor at the Computer Engineering Dept.

Mark Ammar Rayes, Senior Manager at Cisco Systems

APPROVED BY THE UNIVERSITY

08/07/06

ABSTRACT

BUILDING SYSTEMS USING PATTERNS: CREATING KNOWLEDGE MAPS

By Huascar A. Sanchez

This thesis highlights, through an exhaustive study, how current approaches that use pattern languages to build systems strive in providing a systematic and cost-effective method to weave patterns together and build systems. To confront this problem, a distinct representation of pattern languages called Knowledge Maps is proposed. Knowledge Maps' formation is driven by the Software Stability Concepts approach.

Throughout this thesis several issues related to patterns classification, composition, traceability, deployment, and development will be debated to justify the utilization of Knowledge Maps. The main objectives are providing patterns' researchers, framework developers, and application developers the means for weaving similar and different types of patterns, understanding the relationships between analysis and design patterns, and identifying the behind the scenes process for weaving these patterns together and building systems.

Table of Contents

Table of Contents	v
List of Tables.....	x
List of Figures	xi
List of Snippets.....	xiv
CHAPTER 1 Introduction.....	1
The Motivation.....	2
The Problem.....	3
The Objective.....	4
Software Stability Concept's Overview	5
Knowledge Maps Overview	7
The Solution.....	10
Knowledge Maps Methodology.....	10
Research Methodology Undertaken	15
Research Verification and Validation	16
Thesis Contributions.....	16
Thesis Stratification.....	17
Thesis Last Remarks	18
CHAPTER 2 Knowledge Maps (KMs) Methodology	19
Knowledge Maps Levels of Abstraction	21
Mapping KMs to Software Stability Concepts and to Patterns World	23
Goals.....	25

Capabilities	27
Knowledge Maps Formation and Stable Architectural Patterns.....	28
Development Scenarios	32
Deployment and V&V.....	34
Last Remarks	36
CHAPTER 3 The Goals: Significance and Identification	37
Goals Significance.....	39
An Example: Simple E-commerce Application.....	41
Dealing with Goals: Extraction and Assessment	43
Extracting the Goals of a Discipline: The Process.....	46
Dealing with Sub-goals	50
Knowledge Maps Goals.....	52
Goal 1	52
Goal 2	55
Goal 3	56
Last Remarks	58
CHAPTER 4 Capabilities: The Heart of Business	59
Workflow Metaphor	61
Dealing with Capabilities: Identification and Assessment.....	62
A Goal with No Capability.....	65
A Capability with No Goal.....	67
Capabilities' Identification Process: An Example	69
Knowledge Maps Capabilities	74

Capability 1	74
Capability 2	76
Capability 3	78
Capability 4	79
Last Remarks	80
CHAPTER 5 Knowledge Maps: A Roadmap for Building Systems	82
Knowledge Maps Representation: Structure, Quality Factors, and Properties.....	84
Knowledge Maps Structure	85
Knowledge Maps Structure's Notation	86
Knowledge Maps Structure's Relationship with UML.....	89
Knowledge Maps Quality Factors.....	93
Knowledge Maps Properties.....	95
Knowledge Maps versus Traditional Pattern Languages	100
Stable Architectural Patterns Samples	103
Architecture 1	104
Architecture 2	106
Last Remarks	109
CHAPTER 6 Development Scenarios: Setting the Stage.....	110
Knowledge Maps Implementation Issues.....	112
Type versus Class within TOP.....	112
Design Type Specification Template	114
The Uses of Contracts in Design Types' Specification.....	116
The Hook Specification Template	117

User Model or Storytellers Specification	119
Packages Overview	120
Components Overview	121
Last Remarks	123
CHAPTER 7 Deployment and V & V Concerns: Quality Factors	124
Deployment of Knowledge Maps	125
The Conceptual Point of View of Deployment	126
The Practical Point of View of Deployment	127
The Quality Factors of Deployment.....	129
Deployment – V&V Process	130
Last Remarks	133
CHAPTER 8 Knowledge Maps Framework: Initial Work.....	134
Frameworks’ Supported Interfaces	134
The Interfaces’ Implementations and Types’ Structures.....	135
Framework’s Construction Heuristics.....	139
The Framework’s Hooking Facility	141
An Example: Creating Associations between BOs and IOs.....	144
Last Remarks	145
CHAPTER 9 Conclusions and Future Work	146
Future Work.....	148
Last Remarks	150
REFERENCES.....	152
APPENDICES.....	158

APPENDIX A. Detailed Templates	158
Knowledge Stable Analysis Pattern Detailed Template.....	158
AnyContext Stable Design Pattern Detailed Template	168
APPENDIX B. Source Code	177
Frameworks' Main Interfaces or Types	177

List of Tables

Table 1. Knowledge Maps versus Pattern Languages	9
Table 2. The Mapping of Knowledge Maps Elements	23
Table 3. The Goals of Knowledge Maps	52
Table 4. The Scenario Format	68
Table 5. The AnyMechanism's Scenarios	73
Table 6. The Capabilities of Knowledge Maps	74
Table 7. Type Specification 1	115
Table 8. Type Specification 2	115
Table 9. Type Specification 3	115
Table 10. The AnyMediaHook's Specification	118
Table 11. The Goals of Knowledge Maps' Deployment	127
Table 12. Common Stakeholders' Quality Factors	128
Table 13. The Quality Factors of a common Target Environment	130
Table 14. The Summary of Interfaces' Implementations	137
Table 15. The EBT's Structure	139
Table 16. The BO's Structure	139
Table 17. The ExtensionPoint's Structure	139
Table 18. The IO's Structure	139
Table 19. The Classes of the Hooking Facility	141

List of Figures

Figure 1. The Software Stability Concepts Approach.....	6
Figure 2. The Representation of the Knowledge Maps	8
Figure 3. The Knowledge Maps Methodology	12
Figure 4. The Knowledge Maps Representation.....	29
Figure 5. The Formation of Knowledge Maps.....	32
Figure 6. The Adaptation of Capabilities via Hooks.....	34
Figure 7. A Simple E-commerce Application.....	42
Figure 8. The High Level Process for Identifying Goals.....	47
Figure 9. The Learning Stable Design Pattern (Taken from Fayad's pattern archive)	53
Figure 10. The Discovery Stable Analysis Pattern (Taken from Fayad's patterns archive)	55
Figure 11. The Knowledge Stable Analysis Pattern (Taken from Fayad's patterns archive)	57
Figure 12. The AnyMechanism Stable Design Pattern	75
Figure 13. The AnyView Stable Design Pattern (Taken from Fayad's patterns archive).77	
Figure 14. The AnyModel Stable Design Pattern (Take from Fayad's patterns archive).78	
Figure 15. The AnyContent Stable Design Pattern (Taken from Fayad's pattern archive)	79
Figure 16. The Knowledge Maps Main Categories	87
Figure 17. The Elements of Knowledge Maps' Categories.....	87
Figure 18. The Associations between the Elements of Knowledge Maps Categories	88
Figure 19. The UML representation of Goals.....	90
Figure 20. The UML Representation of Capabilities	90

Figure 21. The Three Layer Representation of Knowledge Maps	92
Figure 22. The Software Stability Concepts' Stratification.....	93
Figure 23. The Achievement of the Scalability Quality	94
Figure 24. The Partitioning of the Modeling Domain.	96
Figure 25. Separating Concerns in the Branding Concept (Sanchez, 2005).....	97
Figure 26. The Knowledge Maps Intersection Property.....	97
Figure 27. Using Branding in Biometric Systems.....	99
Figure 28. Using Branding in Marketing	99
Figure 29. The Stable Model-View-Mapping Architectural Pattern.....	104
Figure 30. The Stable MRI Architectural Pattern	107
Figure 31. The AnyContent Stable Design Pattern's User Model	119
Figure 32. The Packages Representation and their Relationships in Knowledge Maps .	120
Figure 33. The Components' Representation in UML	122
Figure 34. The Frameworks' Interfaces or Types	134
Figure 35. The Frameworks' Interfaces Implementations.....	136
Figure 36. The Frameworks' Additional Interfaces Implementation.....	137
Figure 37. The Frameworks' Complete Class Diagram	138
Figure 38. The Hooking Facility's Process Overview	142
Figure 39. The Structure of the Knowledge Stable Analysis Pattern.....	161
Figure 40. The CRC Cards of the Knowledge Stable Analysis Pattern	163
Figure 41. The Behavioral Model of the Knowledge Stable Analysis Pattern.....	164
Figure 42. The Autonomic Computing Context Scenario	166
Figure 43. The CRM System Scenario	167
Figure 44. The Structure of the AnyContext Stable Design Pattern	171

Figure 45. The CRC Cards of the AnyContext Stable Design Pattern.....	172
Figure 46. The Behavioral Model of the AnyContext Stable Design Pattern	173
Figure 47. The Context Sensitivity Customization in CBS Scenario.....	175
Figure 48. The Context-based Storage System Scenario.....	176

List of Snippets

Snippet 1. BO Type Specification.....	113
Snippet 2. AnyMedia's BO Type Implementation.....	113
Snippet 3. Including Contracts in the AnyMedia Class.....	117
Snippet 4. Packages Representation	121
Snippet 5. Creating a BO Sub-class	142
Snippet 6. Dynamically Loading Modified Classes	142
Snippet 7. Associating IOs and BOs	143
Snippet 8. Aggregating an IO to a BO.....	143
Snippet 9. The Input Code	144
Snippet 10. The Resulting Hook	144
Snippet 11. EBT Type	177
Snippet 12. BO Type	178
Snippet 13. ExtensionPoint Type	180
Snippet 14. The IO Type.....	180
Snippet 15. The Analysis Type	181
Snippet 16. The Design Type.....	182

CHAPTER 1 Introduction

Today, the increase in software size, complexity, and cost goes together with the increase in the need for insightful techniques to reduce software development time and complexity. More companies are now trying to develop their software products in less time and with less cost while still maintaining a high quality product. In fact, in the last decade, software patterns have emerged as the magical recipe to achieve these goals and develop high quality software products (Gamma, Helm, Johnson & Vlissides, 1995; Schmidt, Fayad & Johnson, 1996). However, the potential of using patterns in developing systems has not been fully realized.

When software patterns first emerged, practitioners foresaw a huge potential for developing and conveying flexible and quality software solutions. The main idea was to embed software products with the ability to adapt to new requirements easily and with no side effects (e.g., bugs) via software patterns. Software patterns are defined as successful solutions to recurring software problems (Coplien, 1996; Schmidt, Fayad & Johnson, 1996). While there have been several successful stories of using software patterns (analysis, and design, etc.) (Buschmann, 1996; Fowler, 1997), it has been difficult to weave them and build systems of patterns. These systems of patterns are simply a set of related patterns, insightfully woven, that communicate architectural knowledge and styles for a set of high-level problems in particular contexts. Along with the appearance of software patterns, the concept of pattern languages arose to try to ease the weaving of software patterns and form systems of patterns. Pattern languages are simply a collection of interrelated patterns (Schmidt, Fayad & Johnson, 1996). These patterns are combined

in any way to create new environments where practitioners can solve specific software problems. Precisely, the concept of pattern languages has crossed over into the software engineering field to describe prior experiences and the processes that stem from these experiences (patterns), in a simple language where patterns are skillfully woven and combined in any way to solve a particular problem. Yet, this process is still done ad-hoc and is not straightforward enough to ease and speed up the software development process.

Thus, the reason for this thesis is driven by three main questions. First, how can software professionals classify, develop, and utilize analysis and design patterns together towards a problem resolution? Second, what is the “behind-the-scenes” language that guides the sewing of patterns as a whole unit? And third, how can software professionals overcome challenges, other than patterns composition problems (e.g., patterns traceability) that can hinder the development of systems of patterns? The inability to answer these questions detrimentally impacts the understanding of how to put patterns in practice, and will therefore make software patterns’ use more complex than it should.

The Motivation

The motivation of this thesis is to answer the aforementioned questions and synthesize the foundations for patterns classification, composition, traceability, and deployment. These foundations will serve the sole purpose of building systems using patterns in a systematic manner. This motivation will be fulfilled by providing two deliverables. Firstly, a set of quality factors to evaluate software patterns’ definition, application, and solution accuracy. Secondly, a new and different representation of what is known as pattern languages. This new representation is called Knowledge Maps.

The thesis's main goal is to define *Knowledge Maps (KMs)* as the groundwork for an insightful classification of the software patterns governing a particular discipline. Knowledge Maps are enduring mirrors of experience and best practices of what a discipline is, why it is necessary, and how it is to be exploited. The driving force of KMs is the Software Stability Concepts approach (Fayad, 2002a, 2002b; Fayad & Altman, 2001; Mahdy & Fayad, 2002). This approach allows the classification of software patterns within Knowledge Maps according to their nature and rationale (e.g., analysis, design).

This thesis demonstrates how KMs overcome the problems experienced in traditional pattern languages. This demonstration is done by surveying a number of examples within the genre of pattern languages and analyzing found commonalities and drawbacks in these forms, maintaining reference to the instances surveyed. The surveyed pattern languages will be compared and contrasted with the Knowledge Maps using already defined quality factors.

The Problem

Building high quality systems is not an easy task, especially when several factors can undermine their quality success, such as cost, time, and lack of systematic approaches. The promise of using software patterns in software development to deal with these factors or obstacles has led software practitioners to believe in the power of pattern languages. They consider pattern languages as the means for constructing complex systems in a constrained environment.

Software patterns, along with pattern languages, have attracted software practitioners for over a decade. They have seen software patterns and pattern languages as promising techniques that ease and accelerate their software development (Appleton; Coplien, 1996; Gamma, Helm, Johnson & Vlissides, 1995; Schmidt, Fayad & Johnson, 1996). However, developing robust software patterns and pattern languages has not reached the expected ease it should have when dealing with determined software problems, such as pattern composition, stability, etc.; instead, software practitioners construct models that lack of some essential qualities that diminish the quality of the system rather than improving it (Wu, Hamza & Fayad, 2003a).

The response to these issues is the introduction of Knowledge Maps (KMs) as a standard means to classify, organize, weave, and deploy groups of software patterns according to their rationales. To classify these software patterns in accordance to their rationale and create Knowledge Maps, the Software Stability Concepts as the main approach is used. The description of this approach will be provided in further sections.

The Objective

This research aims to achieve a knowledge synthesis for building systems using patterns: creating Knowledge Maps. It is intended to highlight, through an extensive study, how current approaches that use pattern languages to build systems strive in providing a systematic and cost-effective way to integrate patterns and create complex systems. To confront this problem, a new and distinct representation of pattern languages called the Knowledge Maps is proposed.

Throughout this thesis, several issues related to pattern classification, composition, traceability, deployment, and development will be debated to justify the utilization of Knowledge Maps. The main objectives are providing patterns' researchers, framework developers, and application developers the means for weaving similar and different types of patterns, understanding the relationships between analysis and design patterns, and identifying the behind the scenes process for weaving these patterns together and building systems.

The next sub-section will describe the approach that drives the Knowledge Maps realization, providing the required semantics, knowledge organization, and understanding.

Software Stability Concept's Overview

The Software Stability Concepts approach classifies the classes of any system into 3 layers of understanding (Fayad, 2002a, 2002b; Fayad & Altman, 2001): the Enduring Business Themes (EBTs) layer, the Business Objects (BOs) layer, and the Industrial Objects (IOs) layer. Each class of any system is assigned to a particular layer based on its nature and level of tangibility.

The Enduring Business Themes (EBTs) represent the specification classes within a problem's understanding. EBTs' nature is entirely conceptual, which means that their structure is internally and externally stable (Fayad & Altman, 2001). Business Objects (BOs) are semi-tangible artifacts that are internally stable and externally adaptable via extension points (e.g., hooks), or existing patterns, such as Gang of Four (*GoF*) patterns (Gamma, Helm, Johnson & Vlissides, 1995), etc. They represent the business rules or process abstractions necessary to carry out a determined EBT – they are, in other words,

the workhorses of the EBTs. Industrial Objects (IOs) are the context-specific classes that will be attached to the core formed by EBTs and BOs. IOs' nature is entire tangible, which means that they are internally and externally unstable. They change proportionally with the occurrence of new business requirements (Fayad, Hamza & Sanchez, 2005).

The Software Stability Concepts approach provides the foundations for extracting domain-neutral core sets or stable patterns. These domain-neutral core sets are not bound to application-specific concerns by any means. Instead, they remain the same whenever they appear, regardless of their context of application. The following image, Figure 1, represents a concise view of the Software Stability Concepts approach (Hamza & Fayad, 2004, pp. 197-208).

Figure 1. The Software Stability Concepts Approach

Knowledge Maps Overview

The Knowledge Maps are a topology of patterns that are driven by the Software Stability Concepts approach and pattern languages techniques (Fayad, 2002a, 2002; Fayad & Altman, 2001; Mahdy & Fayad, 2002; Salingaros; Wu, Hamza & Fayad, 2003). They consist of knowledge core sets or stable patterns that host the pertinent features and functionality of a particular domain. Furthermore, they can be utilized as the foundations for building other topologies of patterns in other domains.

Building Knowledge Maps for a determined discipline involves numerous skills, knowledge and steps beyond the identification of tangible artifacts bound to a specific context of applicability. It requires a systematic capture and full understanding of the problem domain. This includes describing the problem domain not from its tangible side, but focusing more on its conceptual side, describing underlying affairs related to the problem, and identifying the elements required to fulfill them.

The Knowledge Maps' ultimate representation is driven by the mantra "divide and conquer," which is applied throughout the Knowledge Maps' structure. For instance, the Knowledge Maps are product of partitioning a particular domain into different levels of granularity, so that each level can be managed with ease. In addition to this, the Knowledge Maps are stratified into five main concerns: analysis concerns (called goals), design concerns (called capabilities), knowledge concerns (goals and capabilities together), development concerns (called development scenarios), and deployment concerns (called deployment scenarios). The relationship between these concerns is represented using the Software Stability Concepts approach's three layers: EBTs, BOs, and IOs. Figure 2 shows the representation of Knowledge Maps.

Figure 2. The Representation of the Knowledge Maps

The concept of pattern languages (Appleton; Buschmann, 1996; Fincher, 1999; Salingaros) is crossing over into the software engineering field to describe software development's experiences and best practices using a coherent language that can be used for both talking about a particular problem and creating new environments from the patterns it conveys. This language works by connecting a collection of patterns as if they were in a narrated story. Each of the patterns in the collection is an insightful way to handle or solve a set of recurrent problems in a particular context (Appleton; Fincher, 1999; Schmidt, Fayad & Johnson, 1996). As a whole, they exhibit both the knowledge that is pertinent to a particular domain and the solutions for a set of recurrent problems within this particular domain.

In this thesis, any existing approach and pattern language representations not driven by the Software Stability Concepts approach will be named as traditional approaches and traditional pattern languages respectively. The approaches driven by the Software Stability Concepts approach will be named as the Knowledge Maps. Throughout this sub-section, both traditional pattern languages and the Knowledge Maps will be briefly compared and contrasted. This study will determine the path taken in this thesis. It will also indicate the benefits of using the Knowledge Maps to ease and accelerate the software development process.

The comparison between these two pattern language representations will be illustrated in Table 1; however, this table only offers a brief description of the generated outcomes. The next sub-section describes the systematic approach for implementing the Knowledge Maps.

Table 1. Knowledge Maps versus Pattern Languages

Knowledge Maps (KMs)	Pattern Languages (PLs)
KMs provide a systematic approach that assures focused software solutions.	PLs lack of indicators/guidelines to determine within context software solutions.
KMs classify patterns according to their rationale (e.g., EBTs, BOs, and IOs).	PLs lack of indicators that determine the rationale of their enclosed patterns.
KMs provide full traceability of their enclosed patterns.	Traceability is lost, especially when dealing with deeper levels of PLs' implementation.
KMs provide full generality of their enclosed patterns.	PLs don't guarantee full generality of their enclosed patterns.
KMs provide enduring solutions. Maintenance is minimal.	PLs are hard to maintain and they struggle in providing enduring solutions.
KMs are easy to understand and use when dealing with determined software problems. Everything is based on Goals, Capabilities, etc.	PLs are hard to use and understand when solving a determined software problem.
KMs distinguish between direct and remote knowledge.	PLs don't distinguish between associate (direct) and remote knowledge.

The Solution

The following sub-sections provide an overview of the distinct pieces that form this thesis. It starts with the description of the methodology that will be used to build patterns' topologies. Then, it proceeds with the employed research methodology that will support the completion of this thesis. Major contributions will be also described.

Knowledge Maps Methodology

From a software perspective, several things of prime importance must be fulfilled before, during, and after investigating any problem. Firstly, it is necessary to select the right approach for understanding the problem. This approach must have a systematic and precise problem analysis process. Secondly, it is necessary to have in place a suitable design process to support analysis outcomes and foresee future project changes. Thirdly, it is necessary to provide a classification of the distinct building blocks that conform to the analysis, design, development, and deployment outcomes. Fourthly it is necessary to use validation and verification (V&V) techniques to evidence the value of the work being done, as well as the accuracy of the integration of KMs' building blocks. And finally, it is necessary to have in place the means for achieving a development cost, time, and effort reduction, as well as a "welcome changes" attitude when dealing with volatile requirements.

In this thesis a suitable approach to address all the aforementioned challenges is presented. This approach consists of the unification of pattern language techniques and the Software Stability Concepts approach (Fayad, 2002a, 2002b; Fayad & Altman, 2001;

Mahdy & Fayad, 2002; Salingaros; Wu, Hamza & Fayad, 2003). This unified approach relies on a rationale-driven view to discover and visualize stable knowledge core sets (patterns) within a particular domain. It also facilitates the means for organizing and relating these patterns according to their nature and rationale. Based on these properties, it will be possible for this unified approach to formulate accurate solutions for numerous contexts that share the same core knowledge (rationale or goals and capabilities).

From a bird's eye perspective, the overall process or methodology for creating Knowledge Maps is described herein. This process involves the identification of five main elements: *Goals or Classification, Capabilities/Properties of a particular discipline, Knowledge Maps formation, Development Scenarios, and finally Solution Deployment*. However, further in the thesis, a more detailed process on how to formulate the Knowledge Maps in a systematic manner will be provided. All these elements involved in the creation of the Knowledge Maps, will constitute a set of distinctive patterns. These patterns will be interconnected to form accurate solutions that will satisfy a determined domain's rationale.

The main reason for this methodology is to guarantee an efficient way to develop systems using a set of similar and different patterns. Briefly, this methodology consists of five main steps. Each step targets the completion of a determined phase in a software development lifecycle. Therefore, each step will generate a set of outcomes that will complement an entire system development process, such as functional requirements, design artifacts, development artifacts, extension points, and non-functional requirements.

Each step, along with all its generated elements, will be verified and validated by using a set of heuristics and best-practices. These heuristics and best-practices will be provided in further chapters. Figure 3 visualizes the KMs' methodology.

Figure 3. The Knowledge Maps Methodology

- 1) The first main step, "Analysis/Goals," is concerned with surfacing the implicit goals hidden within a particular discipline. These goals are known as Enduring Business Themes (EBTs) (Fayad & Altman, 2001;

- Hamza & Fayad, 2004, pp. 197-208). This step requires a full understanding of the context wherein the desired solution would be used. This includes describing the goals not from their tangible side, but focusing more on their conceptual side. The final outcome is referred to as Stable Analysis Patterns (SAPs) (Hamza, 2002; Hamza & Fayad, 2002).
- 2) The second main step, “Design/Capabilities,” emphasizes the discovery of the recipes required to fulfill the goals of a particular domain. These recipes are referred to as Business Objects (BOs) (Fayad, 2002a, 2002b; Fayad & Altman, 2001). Without these recipes, there will be a vague understanding (almost none) on how the domain’s found goals will be achieved. The final outcome is referred to as Stable Design Patterns (SDPs) (Chen, Hamza & Fayad, 2005, p. 592 – 596; Fayad & Kilaru, 2005, p. 108 – 115).
 - 3) The third main step is “Forming the Knowledge Maps.” Intuition and experiences from practitioners (e.g., analysts, designers, and developers) support the formation of the KMs. First, practitioners must be aware of the environment wherein the problem is happening. Second, practitioners must examine the goals and capabilities required to describe that environment (e.g., the solution within the context). Third, after acknowledging the environment, goals and capabilities, practitioners must then create synergies between these elements and form stable architectural patterns that will specifically handle a problem of interest. In short, the main goal of this step is to create knowledge core sets from goals and

capabilities. These created knowledge core sets will facilitate the selection of distinct routes/paths out of the available goals and capabilities. Each selected route will satisfy a distinct need in a particular domain.

- 4) The fourth step, “Development Scenarios,” provides essential qualities to a future software solution, such as scalability, traceability, maintainability, stability, and return of investment (ROI). These qualities are inherited from the application of the Software Stability Concepts approach to a problem domain. This step is concerned with how the knowledge core sets will be adapted to specific contexts based upon the utilization of tangible artifacts or IOs. Such an adaptation is possible via the use of extension points. These extension points will attach IOs to the core. By doing this, an unprecedented flexibility will arise, enabling businesses to add, remove, update, and extend functionality from their systems on a real-time basis.
- 5) The last main, fifth, step, “Deployment,” deals not only with how a particular solution and its enclosed knowledge core sets would be deployed in a particular domain, but also with the representation of the artifacts or domain-specific patterns that will aid the deployment process. This includes Stable Analysis Patterns, Stable Design Patterns, and their formed application (IOs) (Hamza & Fayad, 2002).

Research Methodology Undertaken

The development of thesis will be conducted in a systematic way. A considerable time and effort will be invested in collecting and organizing fundamental information related to pattern languages, and other forms of patterns organization and collections. This strategy includes acquiring books that reflect up-to-date software patterns/pattern language techniques – usage and addressed problem, production and deployment processes commonly used in industry – current limitations and future trends related to this area of study, and much more.

This strategy also includes accessing online libraries, looking for software patterns conference proceedings, white papers, published research papers, and journals. Additionally, the trial of currently available commercial software, such as Eclipse and Rational XDE, which use patterns to support software development, will be also part of the strategic input. In doing so, the knowledge behind their usage and capabilities currently offered will be explored. Such a task is complicated because most of these commercial or non-commercial systems don't provide their source code.

After collecting and understanding information related to software patterns, pattern languages, their underlying techniques and ad-hoc classifications, the next step is to introduce a new approach called the Knowledge Maps. Throughout this thesis, the approach's structure, semantics, quality factors, properties, and promising benefits to the software development activities will be described.

After completing these steps, a set of practical examples will be provided to demonstrate the benefits and semantics of KMs. These examples show K Ms' benefits in terms of patterns discovery, organization, classification, development, and deployment.

Research Verification and Validation

This work will be verified and validated via the means of two factors: the utilization of a set of applicability scenarios that implies commonly software patterns/pattern languages usage and expected outcomes, and the validation of the sources of information utilized in this work.

The determined number of scenarios per pattern to be considered are defined based on the proof of validity of current patterns: three applicability scenarios to prove pattern accuracy and correctness. Thus, practitioners will be able to not only visualize the rationale and its associated capabilities, but also to perfectly aid in the development of context-specific aspects.

Thesis Contributions

The section will represent each one of the contributions expected in this thesis. These contributions will be listed according to the problem they address. This thesis will:

- 1) Provide a knowledge synthesis for building systems using the Knowledge Maps.
- 2) Provide a framework for creating the Knowledge Maps, along with some of the Knowledge Maps' qualities, such as scalability, and adaptability. Source code is included.
- 3) Provide the Knowledge Maps' representation to ease the understanding of the path of execution for all the provided stable patterns, as well as their intent and what part of the knowledge infrastructure they represent.

- 4) Document some of the stable patterns provided in this thesis via the use of a noteworthy documentation template.
- 5) Provide at least one sample application that illustrates the actual applicability of the Knowledge Maps' framework provided in this thesis.
- 6) Provide some stable patterns implementation's heuristics to ease the development of stable patterns.

Thesis Stratification

This thesis is stratified as follows. Chapter 1 provides an overview of the entire thesis. It sets the stage for the thesis's proper development. Chapter 2 examines the methodology for forming cost-effective patterns' topologies. Chapter 3 explores the term "Goals" and its important place in the formation of the Knowledge Maps. A goal without a capability is not a goal; therefore, in Chapter 4 a detailed description of the capability element is provided. This also includes a description of what its role is in the formation of the Knowledge Maps. Chapter 5 details what the Knowledge Maps are and their role in the understanding and mastering of any discipline of interest. Chapter 6 concentrates on the formation of development scenarios, especially in the identification of context-specific classes (IOs) and how they are attached to the core formed by goals and capabilities. Chapter 7 provides some insights about the Knowledge Maps deployment, while Chapter 8 provides a description of the framework that will be used to create them. Chapter 9 concentrates on the thesis' last remarks, where a summary of the work that has been done throughout the thesis and the work that will be done in the future. This includes a description of the author's personal contributions.

Thesis Last Remarks

The objectives of Chapter 1 were to introduce the Knowledge Maps or patterns' topology as the means for developing software systems in a cost-effective manner, to show the Knowledge Maps' superiority over traditional pattern languages techniques, and to specify how the rest of thesis will flow. This includes a brief introduction of the content that will appear in each chapter. Chapter 1 also provides a brief description of the Knowledge Maps' structure and properties, along with their benefits, challenges, and constraints.

CHAPTER 2 Knowledge Maps (KMs) Methodology

The first chapter briefly touched the surface of the knowledge required to create Knowledge Maps. This includes a description of the main building blocks, along with the approach that systematically integrates these building blocks to develop software products. Chapter 2 attempts to give more information and tidbits of advice to support the creation of the Knowledge Maps and ease the process of creating software products. In this instance, a profound exploration of the significant elements that compose this approach will be performed.

The purpose of this exploration is to uncover the rules of the game, rules that software practitioners must follow to ease their software development activity. These rules are represented via knowledge core sets or stable patterns. These core sets are allocated into the Knowledge Maps according to their purpose and nature (e.g., analysis, design, development, and deployment). In comparison to the traditional software development lifecycle, the Knowledge Maps cover five concerns: analysis concerns (AC), design concerns (DC), knowledge concerns (KC), development concerns (application concerns), and deployment concerns. Each concern insulates the core sets significant to a domain in question, and suppresses the ones that are unimportant or too application-specific. For instance, analysis concerns deal with requirements' elicitation and understanding (the problem space), while design concerns concentrates on handling the defined requirements (the solution space). Knowledge concerns are the mirrors of experience achieved by creating synergies between analysis concerns and design concerns; they convey architectural knowledge or styles. Development concerns deal

with the formation of the application contexts to be used in the Knowledge Maps' implementation. Deployment concerns deal with the quality factors that must be fulfilled when deploying the Knowledge Maps (e.g., performance, scalability, and adaptability).

The process of extracting the patterns pertinent to the domain under discourse and omitting the unimportant ones is nothing different than the traditional abstraction process. Abstraction refers to the process of extracting the significant details of an entity, or group of entities, and suppressing the unimportant ones (Berard). In the Knowledge Maps, this abstraction process, along with its enclosed levels and participants (e.g., patterns), has been restrained by certain rules or boundaries. These rules and boundaries are in place to ease the understanding and use of the abstraction process in the Knowledge Maps.

Moving forward to the definition of these rules seems to bring along the feeling that creating Knowledge Maps requires all sorts of complex processes, as well as an extensive learning curve. Even though this activity is perceived as a complex activity, it is not. However, to get it right, it will require some dedication and practice from practitioners. To systematically describe the process of creating Knowledge Maps, along with the rules that guide their creation, it will be useful to start with a description of the elements or building blocks that constitute a patterns' topology, and then proceed with its creation.

The Knowledge Maps consist of goals, capabilities, and transient aspects. These artifacts are insightfully woven to specify the groundwork for any discipline of interest. Goals are the essential concerns that determine the rationale of an explored problem of interest. The capabilities are twofold: they are defined as the workhorses of the goals, and they are the processes abstractions required to accomplish a goal. They are internally

enduring and externally adaptable via extension points (Fayad, Hamza & Sanchez, 2005). Transient aspects are those requirement-centric classes that create the application context for goals and capabilities as a whole unit.

Before examining the elements pertinent to the Knowledge Maps, it is important to describe their abstraction process first. After that, software practitioners will be able to understand why these elements are so important for the Knowledge Maps' creation and understanding. To aid this creation and understanding, an intuitive visual representation of the Knowledge Maps is provided. This visual representation will allow software practitioners to understand the underlying language that helps Knowledge Maps' elements integration.

Knowledge Maps Levels of Abstraction

Abstraction is a complex concept that leads its users to confusion and antagonism, since it can be used interchangeably as either a process or an entity. As a process, abstraction concentrates on the isolation of the significant aspects from a discipline while overlooking the less important ones. As an entity, abstraction definition takes a different meaning. It is defined as the representation of a significant part of an aspect of interest via a view, a model, or some sort of focused representation (Berard). On one hand, a model is a simplified construct of a complex entity at hand that is used to enable the reasoning about the relevant elements that form this entity of interest. On the other hand, a view is merely a simplified representation of a model. In other words, both model and view represent the outcomes of an abstraction process. This includes a representation of the different levels of abstraction that a problem can take.

The Knowledge Maps' representation strongly relies on the application of certain levels of abstraction that drives the conception of its building blocks. These levels of abstraction vary from higher levels, where practitioners are involved and aware of one or more aspects that represent a domain's rationale, to lower levels, where practitioners take Knowledge Maps' abstraction one step further, investigating a single aspect as a domain of interest and its representation as a view or model. The application of these abstraction's levels can continue forever, bringing a set of details that does not have anything to do with the Knowledge Maps' context per se. To prevent the occurrence of these irrelevant details, each element pertinent to the KMs has to be exploited using two levels of abstraction: atomic level and non-atomic level.

Aspects in an atomic level represent business-centric classes that cannot be decomposed into several classes. They define the constructs of the problem core. They are represented as the same classes used in any traditional OO class diagram (Hamza & Fayad, 2003), but with a significant business focus. These classes are identified by using two stereotypes: EBT and BO stereotypes. Aspects in a non-atomic level represent business-centric patterns that present a second level of abstraction. This second level of abstraction consists of other classes, and in some rare cases, other patterns. These aspects are identified by using two stereotypes: Pattern-EBT and Pattern-BO stereotypes. The next sub-section will provide more details about the elements that these stereotypes represent.

Mapping KMs to Software Stability Concepts and to Patterns World

Before getting deeper into the thesis, it will be useful to describe the tight relationship between the Knowledge Maps' building blocks, the Software Stability Concepts approach's artifacts, and the terms used in the patterns' world. The following table, Table 2, briefly describes such a relationship.

Table 2. The Mapping of Knowledge Maps Elements

Knowledge Maps	Stability	Patterns
Goals	EBTs	Stable Analysis Patterns (SAPs)
Capabilities	BOs	Stable Design Patterns (SDPs)
Synergy of Goals and Capabilities	EBTs + BOs	Knowledge Map (KM) and many Stable Architectural Patterns (SARPs)
Development Scenario	IOs	Process Patterns (PPs)
Deployment	EBTs + BOs	SAPs, DPs, SARPs
-	-	-
Dynamic Analysis/ The Business Language.	Stability Model/ SW Development in one short.	Building Systems of Patterns

In the KMs' world, everything is classified in terms of goals, capabilities, and transient aspects. These aspects, however, are directly mapped into other fields of study, as in the case of the Software Stability Concepts approach and in the patterns' world. For example, in Table 2, KMs' goals are directly mapped into the Software Stability Concepts approach as Enduring Business Themes (EBTs), since they represent a domain-independent knowledge that contains enduring contracts or rules under which a particular concept is applied. Due to the enduring and reusable quality of goals, along with their conceptual nature, goals can be also directly mapped into the patterns' world as Stable Analysis Patterns (SAPs). Capabilities are directly mapped into the Software Stability Concepts approach as the Business Objects (BOs), since their nature is also enduring and reusable. Due to their properties, they also form the basis for a pattern representation.

Therefore, in the patterns' world, they are known as Stable Design Patterns (SDPs). Goals and capabilities depend on each other. In other words, a goal must have a relationship with one or more capabilities. The same principle applies to capabilities; a capability must have a goal to fulfill. If two or more goals are combined with two or more capabilities, one will end up with a knowledge concern. A knowledge concern is directly mapped into the Software Stability Concepts approach as the synergy between EBTs and BOs. Since knowledge concerns consist of synergies between two or more goals and two or more capabilities, their representation in the patterns' world is known as Stable Architectural Patterns (SArcPs). Due to KMs' stable, adaptable, and reusable nature, they serve as a roadmap for conveying architectural styles that can adapt to new requirements via extension points. These extension points indicate not only how the KMs will be used, but also what the application context of deployment is. Regarding the transient classes, they are mapped into the Software Stability Concepts approach as Industrial Objects (IOs). In the patterns' world, they are known as process patterns.

One important remark about this mapping activity is that regardless of each element's different name, all of them share the same characteristics, values, purposes, and behavior. As a consequence, their names will be interchangeably used in the thesis. The rationale of this mapping activity is to bridge the existing communication gaps between technical and business people and provide a common language. This means that a non-technical person (a manager, for example) can understand or exercise control over the processes that are in place in a particular software product in the same way as a developer can. This is because they now speak the same language.

In the world of the Software Stability Concepts approach, the enduring and reusable qualities of EBTs and BOs are determined by examining the underlying knowledge, sometimes overlooked or assumed by practitioners, found in business issues and rules. EBTs and BOs altogether represent, then, a set of norms on how to understand and solve a set of recurrent problems that require attention from practitioners. As indicated in Table 2, goals and capabilities share the same vision as the EBTs and BOs. They all are business-centric and within-context aspects that provide a retrospective of a discipline's rationale. The process of identifying the EBTs and the BOs of a problem can be explored in detail in (Fayad, 2002a, 2002b; Fayad & Altman, 2001; Hamza & Fayad, 2003).

The following sections describe one important element of the Knowledge Maps: the Goals. The definition of this element is vital during the construction of a patterns' topology for a determined discipline, since it specifies the fundamental themes that drive the understanding of this discipline. Therefore, practitioners must pay greatly attention to them, especially when building their own patterns' topologies.

Goals

From a KMs' standpoint, goals are business-centric themes that provide an enduring retrospective of a discipline's rationale. They represent the essential themes of any discipline; themes that are free of irrelevant details not pertinent to the discipline under consideration. For example, the concept of "Friendship" is a universal theme applicable across different cultures, fields, and beliefs. Basically, this concept is not bound to a specific application context.

Generally speaking, goals are difficult to discover since they are off-content themes hidden within the complexity and lack of understanding of a problem of interest. For example, imagine being involved in a project that requires the development of a scalable biometric system for a top-security company. As a developer, you will automatically suggest certain objects you think will represent the solution for the desired system. These objects may be: Suspect, SystemOperator, FacialScanning, FaceImage, Fingerprints, RetinalPattern, IrisPattern, DNASquence, SuspectDetection, CameraSensor, FacialShape, Database, etc. As you can see, these objects are specific to a particular context. Therefore, they cannot be the focus of your solution, especially since they will change proportionally to the introduction of new technologies and subsequent requirement changes. Therefore, if you include them as part of your solution for the biometric system, there is a big chance that you will lose the money, the time, and the effort invested in this development due to the occurrence of new requirements.

In order to overcome these problems, it is necessary to start thinking in terms of goals, because it is the pervasive nature of goals that will help software products to survive constant changes in requirements. Using the biometric system example, but now focusing on goals, one will be able to generate a stable solution. One needs to start with the definition of the area of interest; that will be biometrics. Biometrics is the science that measures the physical or behavioral features of an entity. This definition, in conjunction with the question “what is biometrics for?” will assist in the process of defining one of biometrics’ goals. This first goal will be then “Branding” (Sanchez, 2005). Branding is defined as the mental construct that creates an association between an entity and its brands by forming a unique identity that differentiates this entity from its

peers (Sanchez, 2005). Branding's peculiar behavior strongly complements the use of biometrics. Therefore, if you want to solve the biometric system project, you must include the "Branding" goal as part of your solution. Branding as a conceptual model consists of the following elements: BrandingType as the type of branding; AnyEntity as the handler for any type of entities; AnyBrand to deal with any type of brands; AnyMechanism to control all possible mechanisms involved in branding; AnyIdentity to represent the sum of all essential qualities that will be used by a brand, and AnyParty, which can be the branding practitioner and the spectator.

In summary, a goal answers the question, "what is the concept for?" However it fails to answer how the concept does this. The latter is handled by the capabilities of the concept. Goals and capabilities depend on each other and provide the foundations for an unimaginable set of architectures. Therefore, before getting into how goals are associated with capabilities, it will be useful to define what capabilities are.

Capabilities

A goal without a capability is not a goal. Capabilities are the business-centric workhorses that support the goal's realization or fulfillment. As are goals, capabilities are enduring artifacts, but with a minor difference: they are externally adaptable via extension points (Fayad, Hamza & Sanchez, 2005). Their adaptable nature can be determined only by examining the relationships between the underlying business and a direct application context. This also includes the application of the right "hooking code" to the underlying business. Firstly, these relationships can be inheritance, aggregation, or associations. Secondly, the "hooking code" is responsible for weaving business and

industry (application) together rather than focusing on the generalization-specialization principle. Hooks create an environment where capabilities are able to attach any transient aspect without changing the capabilities' internal structure.

Capabilities are less hard to find than the underlying goals of a discipline, especially since they tend to represent any knowledge skill, process, and/or ability associated with the execution of a specified course of action or action workflow. The discovery of capabilities is assisted by answering a few questions. For instance, how can a software practitioner approach an underlying goal? What does a software practitioner need to fulfill it? Who is it going to use it? As an example, imagine that the goal of interest is the "Sampling" concept (Sanchez, Lai & Fayad, 2003). By asking the above questions, a software practitioner will end up with the following classes: SamplingEntity, SamplingType, Applicability, AnyMechanism, AnyCriteria, AnyMedia, and AnyParty.

Capabilities are adaptive aspects that assure a reduced cycle time for coping with a vast number of transient requirements, handling other goals and capabilities in KMs. This peculiarity of capabilities allows both an on-demand adaptation and flexibility to transient aspects and an on-demand scalability of the environment, if necessary, to achieve a goal. This particular behavior will assure a faster return of investment (ROI), as well as a quality solution for a determined problem domain.

Knowledge Maps Formation and Stable Architectural Patterns

In Chapter 1, a visual representation of the Knowledge Maps was provided. This visual representation was provided to give an idea of how KMs were structured. From a positioning level, that representation would be enough; however, from a research level,

which is this work's main objective, such a representation is not enough. Therefore, a more technical representation is required. Figure 4 shows such a representation. This representation consists of different components, such as Goals, Capabilities, Knowledge Concerns, Hook Facility, Development Scenarios, and Deployment Concerns.

Figure 4. The Knowledge Maps Representation

As illustrated in Figure 4, the Knowledge Maps reach far beyond an organized visual representation. Underneath their images and symbols lie a set of well-elaborated pieces of code (e.g., Java code) that determines how goals, capabilities, and development scenarios, etc. are to be represented in terms of Java classes. In further chapters, software practitioners will be able to read how these building blocks are implemented in Java. Java has been selected as the target programming language to be used in this thesis. It

has been selected due to it is a fully object-oriented programming language. Software practitioners can also use other programming languages, such as C#, VB.NET, C++, and AspectJ, etc., if they so desire.

The Knowledge Maps' creation requires skills, knowledge and steps beyond the identification of tangible aspects bound to specific contexts of applicability. It requires a systematic capture and full understanding of the domain in which it resides. This includes describing the domain not from its tangible side, but focusing more on its conceptual side, describing its underlying affairs or essentials, and describing the elements required to fulfill these underlying affairs.

The process consists of seven steps:

- 1) Domain Knowledge Partitioning. This is an interactive process where a particular domain or discipline is decomposed into distinct levels of abstraction or granularity. By doing this, it will be possible for a software practitioner to manage a domain's knowledge with ease.
- 2) Single Domain Focus. This step concentrates on selecting a sub-domain (a product of the domain partitioning) of interest at a certain time (this sub-domain will be also represented as a topology of patterns). After that selection, software practitioners can start extracting the goals that drive this sub-domain's rationale. This step can be applied to all selected sub-domains.
- 3) Place Goals. This step focuses on placing the extracted goals into the Knowledge Maps' goals section (3-5 Goals per KM).

- 4) Identify Goals' Capabilities. The focus on this step is to identify, filter, and evaluate potential capabilities that will fulfill the goals that were identified in the patterns' topology.
- 5) Connect Capabilities to their Goals. The focus of this step is to specify how a selected domain's rationale will be fulfilled.
- 6) Branch out to other Knowledge Maps (KMs). This step is twofold:
 - a. Because some partitioned two or more sub-domains' KMs can have overlapping capabilities, it is possible to connect two or more Knowledge Maps that were once part of the same domain before their partitioning.
 - b. One discipline's partitioned sub-domain can be associated with another discipline's sub-domain (remote knowledge). This remote knowledge serves as both a usage indicator of the current domain knowledge and a position indicator of the sub-domain being explored.
- 7) Knowledge Maps' formation. After the completion of the above steps, the final result will be a set of patterns' topologies that will realize the rationale of particular domains. The number of patterns' topologies will depend upon how deeply a particular domain is partitioned.

The results of these steps will be a set of interrelated goals and capabilities that serve a particular domain's purpose. Figure 5 illustrates this process.

Figure 5. The Formation of Knowledge Maps

The next section describes the Knowledge Maps' development scenarios and how they are attached to the core formed by goals and capabilities.

Development Scenarios

Development scenarios are determined by examining how BOs cope with determined application contexts, full of IOs, by using extension points or some sort of "hooking code." Application contexts tend to be volatile since they are driven by today's business and cultural responses, not future ones.

Referencing the example where it was required to develop a biometric system. One can perceive how developers overlooked the sometimes “off-content” essence of a problem and selected details of this problem with which they were familiar. This prompt response ended up with certain classes that were internally and externally unstable: FaceScanning, FaceImage, FacialShape, Database, Suspect, Operator, etc. Within the KMs’ standpoint, these unstable classes are known as “Industrial Objects.”

The process of finding these classes is common and very straightforward. One way is to follow traditional methods such as the Abbot’s approach (Abbot, 1983, pp. 882-894) or other methods where the target is the finding of verbs as candidate classes. Another way is to first examine the capabilities and directly map them to the physical world. Then, one can apply the just mentioned software engineering methods and find the classes that complement each mapped capability. This process may not sound earth-shattering, but it is effective, especially when dealing with a vast number of problems, each one different in nature, at hand, and a reduced notion of the application context under discourse exists. The beauty of this process lies in the utilization of these transient aspects or IOs, via extension points or hooks, to complement the performance of certain capabilities without changing these capabilities’ internal structure.

Figure 6 illustrates the aforementioned process. This image illustrates how capabilities are aware of their goals, how they are interconnected, and how they are adapted to application contexts via extension points.

Figure 6. The Adaptation of Capabilities via Hooks

The next section describes the last element in the Knowledge Maps: the deployment and V&V concerns.

Deployment and V&V

Deployment and V&V focus on the definition of robust Knowledge Maps by using a set of quality factors pertinent to a discipline of interest. Providing a definition of these robust KMs is a challenge, especially when dealing with disciplines that have unique characteristics and behavior, and a lack of complete and systematic processes to support their perusal.

The deployment of KMs is intended to serve as a mirror of a collaborative experience gained over the definition of goals and capabilities. This collaborative experience is translated as the quality factors governing the deployment of KMs.

These quality factors are identified by following the same process used to identify goals. In fact, quality factors are also goals of a discipline, but with a central focus on how this discipline is used (and how it should be used) in large or more specific domains.

These quality factors usually represent non-functional requirements, such as reliability, performance, scalability, traceability, and usability. However, the definition of these quality factors is highly dependent upon the discipline or subject of interest. Due to the reusable and domain-independent nature of goals and quality factors, some cases may exist wherein some of the quality factors of one discipline appear in another one, and vice-versa. But this is not always the case.

Clearly, to make knowledge accessible, complete, and accurate, it is important to explain or note how the context is set and what the purpose of its use is. One way to represent this usage notion is to identify the quality factors pertinent to a discipline and indicate which capabilities are required to complete them (this also includes the attachment of a set of Industrial Objects). Along with the synergy between goals and capabilities, the synergy between quality factors and capabilities provide the basis for the generation of an unimaginable number of software architectures. This particular behavior is illustrated in Figure 6. In the end, it is recommended for the software practitioners to apply the same heuristics and assessment indicators that were used for goals identification and V&V, if they want to end up with a set of stable software architectures.

In general, the rationale of providing a deployment concern in the KMs guarantees the proper forming and usage of core sets and the enclosed patterns pertinent to a discipline of interest. Further chapters will expand upon the Knowledge Maps idea, along with its development.

Last Remarks

The main objective of this chapter was to describe in detail the structure of the Knowledge Maps as well as the distinct concerns it develops. These concerns are analysis concerns, design concerns, knowledge concerns, development concerns, and deployment concerns. These objectives were implemented by outlining the essence of the KMs in a proper order. Further chapters will provide more insights on the elements that form the KMs.

CHAPTER 3 The Goals: Significance and Identification

Goals represent what a concept is for, but not how the concept does it (capabilities). They convey the enduring business rules under which the capabilities of a concept must function. Goals are an important piece of Knowledge Maps formation since they are a function of the discipline's rationale awareness and a within-context analysis. By discovering goals, there will be a focused understanding of a subject matter or problem domain. This focused understanding will lead software practitioners to an accurate problem resolution.

The process of identifying goals in the analysis phase to describe software requirements is not new; it has been addressed before (Anton, 1996; Anton & Potts, 1998; van Lamsweerde, 2001). However, the method being introduced in this thesis assesses the definition and identification of goals at a much higher level of understanding. For instance, previous approaches suggest that goals are individual software functions (activities) and/or constraints that may experience certain refinement over time due to changes in organizational processes (Anton, 1996; Anton & Potts, 1998). Usually, the number of these functions can be large and overwhelming to handle, especially if the processes documentation is exhaustively analyzed (Anton, 1996; Anton & Potts, 1998).

This thesis shares a different view from others about what goals are. In this view, goals are themes that determine the overall rationale of a software solution. These themes are enduring, and they are neither functions nor constraints related to the concepts' transient matters as they have been suggested in previous work (Anton, 1996; Anton & Potts, 1998).

Goals' presence becomes obvious in the analysis phase when developers deal with business people and/or stakeholders. Business people tend to think about a problem in terms of principles that describe what the problem is for, while developers tend to think in terms of the elements that can be adapted, or even created, to satisfy a process. Principles, by nature, are more enduring than processes. In this instance, the branding concept will be used as an example to demonstrate this behavior. Branding is an enduring principle that has been part of people's lives throughout time. The earliest shepherds branded their cattle to identify them and differentiate them from other shepherds' cattle using a set of machinery, such as iron brands and paint tars. Today, branding is used in other fields such as marketing and HCI. The process to achieve branding is totally different in each field, but the principles of "what branding is for" remain the same, which are the identification and differentiation of an entity among its peers. This still does not sound earth-shattering and seems to be irrelevant to the analysis purpose; however, throughout this chapter, software practitioners will perceive how important it is. From a business perspective, it is important to remember that any type of business is driven by enduring principles that determine its function and success. Unfortunately, many companies overlook these principles and depend on available machinery to temporarily extend their businesses' lives. This is a viscous cycle that will incur more cost, time, and effort to these companies. This situation shows how important is to focus on those stable aspects in a business than to focus on proportionally patching existing deficiencies with new machinery to the appearance of new requirements.

The terms principles, essential themes, or enduring themes all refer to the same concept: the goals. In the rest of the chapter, software practitioners will be able to read on why goals are important elements of the problem space, and what their identification process is. Later in the chapter, some of the essential themes driving the Knowledge Maps' realization and understanding will be introduced. These essential themes will be described using a short pattern documentation template. Two complete pattern documentation templates will be provided in Appendix A.

Goals Significance

Goals have always been part of OO Analysis or problem space; however, their use was always implicit and, in several cases, ignored by software practitioners (van Lamsweerde, 2001). For instance, software practitioners usually put more effort into trying to solve a problem than trying to understand it first. Therefore, requirements were never synchronized with the machinery or elements used to fulfill them. As a consequence, businesses experienced great losses due to the software was not completed on time, or the developed software was simply the wrong one. The use of goals to overcome these problems is starting to be recognized (Anton, 1996; Anton & Potts, 1998), especially during the requirements acquisition phase. Software practitioners are starting to think in terms of "why do they need this subject (system)?" or "what are the objectives that they are planning to achieve with this subject?"

The problem with the above approach is that practitioners have a tendency to think in terms of the application's elements that are most likely to change over time due to the occurrence of new problems or new business requirements (Fayad, Hamza &

Sanchez, 2005). The approach being introduced in this thesis concentrates on those goals that will remain stable over time. These goals will overcome application changes, and new business requirements, etc., since they are created at a knowledge level and not at an application level.

Axel van Lamsweerde provides in (van Lamsweerde, 2001) a set of reasons why the definition of goals is so important to the requirements elicitation process. Knowledge Maps include their reasons too. The ones that relate to Knowledge Maps are product of certain properties of goals. These properties are (Hamza & Fayad, 2003) (some properties were omitted since they merely apply to a model that consists of goals and capabilities):

- **Stable.** A goal must represent a stable and conceptual structure that determines a subject matter's rationale.
- **Natural.** It is important to present these goals, in a cohesive and natural, no less important, language to assure their reusability elsewhere.
- **Domain-Independent.** A goal must represent a conceptual structure that appears in multiple application domains.
- **Single EBT.** A goal must represent a single EBT. This means that there a focus on one problem at a time.

These properties were described to help practitioners understand the reasons why goals are important. These reasons are:

- 1) **Goals' definition implies a completeness of the requirements specification.**
A requirement specification was said to be complete if a determined goal was achieved.

- 2) Goals are a retrospective of the subject's rationale. This retrospective can be understood and applied by management and technical personnel.
- 3) Goals facilitate a natural mechanism that allows management and technical groups to be on the same page with respect to what the subject (system) is for.
- 4) Goals' definition is a focused process – irrelevant details are avoided.
- 5) Goals are enduring themes that are not bound to volatile information (e.g., process documentation).
- 6) Goals' proper identification will drive the discovery of their capabilities. In other words, once a software practitioner has found (and evaluated) a determined goal, he/she will be able to determine the goal's capabilities.
- 7) Goals' identification process facilitates great team dynamics. Since goals are described in a simple and natural language, they are clearly understood by management and technical personnel. Therefore, managers and technical staff can actively participate in the identification of goals.

An Example: Simple E-commerce Application

To illustrate why goals are important, a simple e-commerce example will be provided. This example will convey not only why goals are needed, but also how they are used to leverage the e-commerce solution.

Imagine that a business firm is requesting JustACompany to develop the firm's new e-commerce application. Without any hesitation, JustACompany accepts the project and it gets ready for the development of it. Now JustACompany's software development

team proceeds with the requirements elicitation process. During this process, the team reviews the problem statement. Then, they will look for the candidate classes of the problem using traditional software engineering approaches (Abbot, 1983, pp. 882-894; Fayad, Hamza & Sanchez, 2005). A typical result will consist of classes such as Customer, ShoppingCart, CreditCard, Database, Catalog, Order, and Product. These objects are usually extracted from a problem statement that was given to JustACompany. The model is illustrated in Figure 7.

Figure 7. A Simple E-commerce Application

Consider the above e-commerce example as a business case. This simple model will have a tendency to be re-done every time new requirements appear on the loop. Basically, proportionally to the occurrence of new business requirements, this e-commerce solution will be susceptible to drastic changes and adaptations to its structure (Fayad, Hamza & Sanchez, 2005). These drastic changes may jeopardize the life of the e-commerce application, from a business perspective, since businesses are usually reluctant to keep financing a project that produces more costs than benefits. Therefore,

this e-commerce application has two possible ends: abandonment and re-engineering. Neither one of those states is positive for the business. In fact, no pleasing effects will arise unless there is a focus on the e-commerce application's business themes that are likely to endure throughout its life. So, JustACompany will end up asking the following questions: what are those enduring themes of e-commerce? Is there a way to capitalize on the declared e-commerce's enduring principles and trace them over the success of the business? How can developers find these enduring themes? To answer these questions, it will be necessary to describe the enduring business's identification process.

These questions will ease the understanding of the e-commerce application's problem space. For instance, the developers involved in this project will be able to characterize and acknowledge the e-commerce subject's scope, nature, and core elements. These core elements, purely conceptual, will provide a roadmap on how the e-commerce application should be developed. The following section will describe the process for identifying the goals of a subject matter or discipline.

Dealing with Goals: Extraction and Assessment

Focusing on goals during the analysis phase is vital, especially if you want to understand the problem space of any subject matter. Instead of focusing on aspects that come and go proportionally with the appearance of new requirements and technology, you will be focusing on elements that will remain stable over time. Within the Knowledge Maps' realm, these goals are classified into three categories: personal goals, business goals, and project goals. This classification will set the boundaries for the identification and assessment of the goals of a determined discipline.

Goals are where businesses, projects, and persons (hosts) meet when trying to understand a particular subject of interest. Goals achievement resides within the harmony or equilibrium between these goals' hosts. This harmony is recognized by the relationships and levels of organization among each concern that forms these hosts' rationale. Each concern may include businesses, projects, and persons' own values, desires, and constraints, such as mission, vision, needs, meaning, ROI, trade-offs, current state, and future destinations. Each concern can be addressed in isolation; however, the relationship among them defines the overall flow or guidance for how these concerns are handled and how they complement with each other.

Business goals are the mission constructs of any business. They define the path that any business has to follow if it wants to succeed. This path consists of high level abstractions that will drive a business performance and focus on a determined society (e.g., a market-driven society). However, since this path consist of high-level abstractions, it is common that business goals are built from other fine-grained (level of detail) goals. For example, the fulfillment of business goals depends on the fulfillment of project goals, and the fulfillment project goals depend on the fulfillment of personal goals, and so on.

Project goals are need-driven concerns formulated to reach a future deliverable for a set of customers (internal or external). They reside in the satisfaction of determined needs, such as project completion, availability, solution scope, and time to market, etc. These needs are then transformed into products (e.g., software applications and/or services). Basically, project goals are just "*the compass guiding the direction of your project*" (NLG Project Planning: A Tutorial). The fulfillment of project goals will place

any business one step closer to achieving its overall business goals. For example, imagine that in your company two of your business's goals are to assure your products *acceptance* by possible customers, and to assure a faster ROI. If you want your company to fulfill these particular goals or needs, you will have to have in place the proper means for assuring a customer *satisfaction*, customer *support* in case a customer has concerns or questions regarding your product, and *efficiency and efficacy* for how to *sell* and *deliver* your products to potential customers.

After acknowledging these project goals, you will have a better idea of what infrastructure is actually needed, what the pre-requisites are to implement it, and what its solution scope is. This may result in the definition of a set of products that will achieve these goals. A Customer Relationship Management (CRM) system may be required, for example. In a situation like that is where the responsibility of technical and business staff come into place to formulate and implement the desired infrastructure. This infrastructure consists of a set of products or services required to assure the completion of business goals.

Dealing with personal goals is a little bit different than dealing with business and project goals. This is because personal goals exclusively deal with the goals of each individual participating in the company or project of interest. These goals can be considered the most delicate goals in a company, especially since they implicitly drive the success of the project goals. Here is where you, as team leader or manager, try to bring each individual on the same boat, while still assuring that each individual's goals are achieved. These goals may include incentives that will motivate individuals, such as multi-cultural environments, team acceptance, and team collaboration. In this case,

management, as well as team leaders, must discern the required steps to create an environment where personal goals and project goals live in harmony, so that the company can accomplish their established goals.

Within a business environment, each goal category can be addressed in isolation. However, it is the harmonized relationship between them that will determine the success of any business. Throughout this thesis the focus will be on project goals, especially the ones related to project development (e.g., products and services). However, it is important to acknowledge the value of personal and business goals in the achievement of project goals. Failing to acknowledge these goals' value will jeopardize the achievement of project goals. Therefore, these goals must not be ignored by all means during any project goals' definition.

In summary, identifying the underlying goals from a discipline is a challenging task for both novice and experienced practitioners. The Software Stability Concept approach (Fayad, 2002a, 2002b; Fayad & Altman, 2001) shows great promise for the identification of these goals. The next section describes the goals identification process in a cohesive, not less important, way, so that one can understand it.

Extracting the Goals of a Discipline: The Process

Through a set of straightforward questions this section will illustrate how the goals of a discipline are identified, assessed, and filtered. The high-level process for extracting these goals is illustrated in Figure 8. One important remark is that this process may be performed in parallel with the capabilities identification process.

Figure 8. The High Level Process for Identifying Goals

Using the previous e-commerce example, the process for identifying, filtering, and evaluating goals will be illustrated. In this instance, the JustACompany business will be able to answer the question related to the e-commerce's enduring themes:

- 1) Selecting the subject of interest. This step is designated to help guide software practitioners to examine the subject of interest. Irrelevant details, such as application details, must be ignored during this subject's exploration. Practitioners must focus only on the aspects that define the core knowledge of e-commerce. This step is supported by asking the following questions:

- a. What is the subject or problem being made? The subject is e-commerce.
 - b. Are you aware of scenarios where the subject (e-commerce) appears? This will help one to narrow down the scope of the subject being explored.
- 2) Applying the Software Stability Concepts approach. This step follows some of the heuristics provided in (Fayad, 2002a, 2002b; Fayad & Altman, 2001; Hamza & Fayad, 2002) to identify an initial list of goals. These heuristics are applied by asking the following questions:
 - a. What is the subject/problem (e-commerce) for? What are the reasons for this subject matter/concept to exist?
 - b. What does the subject (e-commerce) do? (See Chapter 4).
- 3) Identifying goals and capabilities. This step's outcome will be a conglomeration of candidate goals, such as ROI, Product Navigation, Product Selection, Trading, Order Handling, Customer Service, Convenience, and Security.
 - a. These are candidate goals because they have been identified by solely examining what is being known or found about a subject in current literature.
 - b. Also, some of these candidate goals may be still referencing the subject matter from an "Industrial Objects" perspective.

4) Filtering the identified goals and capabilities. This step concentrates on filtering the identified list of candidate goals. By filtering this list, there is a chance that one will end up with the right goals. This step is driven by the following questions:

- a. Does a goal depend on adjacent goals to exist? If so, this goal must be removed from the list of candidate goals. Is a goal part of a stakeholder's wish list? If so, this goal must be removed from the list of candidate goals. A "wish list" is defined as the list of goals that one may like to achieve or have at certain point of time.
- b. The resulting list of goals will be smaller than the initial list. The new list of goals will include Trading, Order Handling, Convenience, and Security.

4.1) Evaluating each identified goal (EBT) using the EBTs' properties.

This step is driven by the following questions and heuristics from (Cline & Girou, 2000, pp. 101-106; Fayad, 2002a, 2002b; Fayad & Altman, 2001; Hamza & Fayad, 2002):

- a. Can one replace any of the remaining goals with another goal? If so, the affected goal must be removed from the existing list of goals.
- b. Is a goal internally and externally enduring? In other words, does a goal reflect the essence of the subject matter's existence? If not, the goal must be removed from the existing list of goals.

- c. Does a goal have a direct physical representation? If so, this goal must be removed from the existing list of goals.
- d. The final list will consist of three goals, Trading, Convenience, and Security.
- e. These goals will determine “what e-commerce is.” Also, it will answer the question “is there a way to capitalize on the declared e-commerce’s enduring principles and trace them over the success of the business?”

4.2) Evaluating each identified capability (BO) against the properties of BOs. See Chapter 4 for more details.

Steps 5, 6, 7, 8, 9, 10, and 11 are addressed in Chapters 4, 5, and 6 respectively.

Regarding the question, “is there a way to capitalize on the declared e-commerce’s enduring principles and trace them all over the success of a business?” the answer is yes, there is a way. This way includes the definition of the goals’ capabilities, and the synergy between the goals and their capabilities without losing generality. Both steps will create the foundations for developing a set of architectural styles on an on-demand basis. These foundations will be the answer the aforementioned question.

Dealing with Sub-goals

Similar to goals, sub-goals are enduring principles that determine the rationale and nature of a discipline or domain. They are also represented as EBTs. However, their existence and satisfaction are not as significant as the role of the main ones. Basically, sub-goals can be considered as ancillary goals that would be nice to have within a

determined event or situation. Therefore, including sub-goals as part of the essential elements of a discipline may bring more problems than actual benefits. Firstly, one is extending the context or boundary of one's study. This extended context will contain additional axioms or constraints and events that may, in some cases, contaminate the direction of one's subject study. Secondly, one is facing certain trade-offs during the designing phase due to the inclusion of sub-goals in a subject's study. This will usually happen if a new need appears during subject's study. For example, will your goals be achieved without satisfying your sub-goals? Please think carefully about this question.

The concept "Marriage" can be used as an example of a goal that contains ancillary goals or sub-goals. These ancillary goals include Love, Friendship, Companionship, and Harmony. If one asks the question "can marriage exist without the satisfaction of any of its sub-goals?" honestly, the answer is yes, marriage can exist per se without the satisfaction of any of its sub-goals. This does not mean, however, that is wrong to satisfy marriage's ancillary goals. What this paragraph tries to tell you is that unless you really need to include these ancillary goals as part of your study, you should not deal with them nor include them as part of your solution.

The next sub-section covers the main goals of the Knowledge Maps. All the main goals are provided in Table 8. However, only three goals will be described using a short template.

Knowledge Maps Goals

This section covers the goals of the Knowledge Maps. They are illustrated as stable analysis patterns (see Chapter 2 for reference). There are a total of 7 goals driving the Knowledge Maps' function and rationale. Table 3 summarizes these goals for a quick reference.

Table 3. The Goals of Knowledge Maps

Goal	Description	Provided?
Learning	In this domain, learning is the cognitive process of acquiring skills or knowledge about a specific discipline.	YES
Discovery	It is defined as the process or act of finding something or somebody unexpectedly or after searching.	YES
Knowledge	In this domain, it portrays the distinct aspects (goals and capabilities) of the Knowledge Maps.	YES
Abstraction	A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.	NO
Visualization	In this domain, it visualizes the existence of certain goals, capabilities, and transient aspects in the KMs, as well as their relationship to one another.	NO
Synthesis	The ability to create new knowledge out of pertinent aspects of a particular domain. This includes the association of direct knowledge and remote knowledge.	NO
Leveraging	This refers to the reuse of source and/or target patterns from one set of core sets in remote patterns' languages specific applications.	NO

Goal 1

- Name. Learning Stable Analysis Pattern (Fayad & Telu, 2005).
- Context. The Learning stable pattern can be applied to various patterns in various day-to-day fields or applications. For example, in a formal learning experience, a student has a syllabus from which he or she is taught and then tested on. In the workplace, people often learn through experience. Generally, a person with a graduate degree and 5 years of

experience in a particular field is more knowledgeable about that field than a student who has just graduated in the same field because, while working in the field, the experienced individual has learned through the process of conquering challenges presented in the workplace.

One can see learning happening as a routine in non-humans too. Birds learn by observation and experience. A baby bird starts by watching its mother or father fly and trying to fly small distances by hopping from one tree to another. As it grows, the bird begins fly better until it masters the talent.

- Problem. How to create a conceptual model for the learning concept that is general enough to be applicable to any domain.
- Solution and Participants.
 - Solution. Figure 9 illustrates this solution.

Figure 9. The Learning Stable Design Pattern (Taken from Fayad's pattern archive)

- Participants.

- Classes.

- Learning. Represents the learning of any party.
This class consists of behaviors and attributes that control the learning process.

- Patterns.

- AnyActor. Represents any actor who gains any kind of skill or achieves learning through various means.
 - AnySkill. Represents the skill that is gained by an actor by adopting the learning process.
 - AnyLearningProcess. This is the methodology that an actor adopts to gain a skill. This can be experience, trial and error, formal education, or just watching.
 - AnySubject. The subject or the topic of learning.
 - AnySource. Represents the source from which this learning process takes place. This can be a book, the Internet, a paper, or a journal. In some cases such as in group discussions, this aspect might not actually exist unless the group discussion is about a book or a journal.

Goal 2

- Name. Discovery Stable Analysis Pattern (Khadpe, 2005).
- Context. This pattern can be used in several applications where a discovery concept is needed. For example, it can be used for the discovery of facts, events, and patterns.
- Problem. Discovery is an enduring concept that spans multiple applications ranging from the discovery of the universe to the discovery of a mathematical formula, etc. How can one generate a model for this concept that is applicable to numerous domains?
- Solution and Participants.
 - Solution. Figure 10 illustrates this solution.

Figure 10. The Discovery Stable Analysis Pattern (Taken from Fayad's patterns archive)

- Participants.
 - Classes.
 - Discovery. Describes the discovery process.
 - Patterns.
 - AnyDiscoveryType. Represents the different types of discoveries in different application areas.
 - AnyDiscoveryMechanism. Represents the Pattern-BO, which deals with different kinds of discovery mechanisms.
 - AnyDiscovery. Represents the Pattern-BO, which represents the desired discovery.
 - AnyActor. Represents the individual(s) who interact with a discovery mechanism, or a scientific group responsible for the discovery process.
 - AnyEvidence. Represents the proof of the discovery.

Goal 3

- Name. Knowledge Stable Analysis Pattern (Telu, 2005).
- Context. Knowledge can be gained through experience or studies. It represents a collection of facts, rules, tips, or lessons learned with respect to anything that must be synthesized to create knowledge. This pattern will be used to represent knowledge synthesis and acquisition.

- Problem. How does one build a model that encloses the common core knowledge of the Knowledge concept?
- Solution and Participants.
 - Solution. Figure 11 illustrates this solution.

Figure 11. The Knowledge Stable Analysis Pattern (Taken from Fayad's patterns archive)

- Participants.
 - Classes.
 - Knowledge. Describes the synthesized knowledge.
 - Patterns.
 - AnySkill. Represents the different types of abilities that an actor can gain via any mechanism.
 - AnyMechanism. Represents the entity that deals with different kinds of mechanisms.
 - AnySubject. Represents the information pertinent to a domain that will be used for future reference.

- **AnyDomain.** Represents the environment that will be explored in search of rules, constraints, and knowledge pertinently related to one or more subjects.
- **AnyActor.** Represents a person or a group of people who is or will be gaining knowledge.

Last Remarks

The main objectives of this chapter are threefold. The first objective was to provide a set of heuristics to ease the process of identifying and assessing the goals of a discipline. The second objective was to provide the basic understanding of what goals are, and why they are needed in the analysis phase of any problem. The third objective was to provide some of the goals that drive the Knowledge Maps' formation. These goals were provided using a short-pattern documentation template. These objectives were implemented by using the Software Stability Concepts approach. By using this approach, it was possible to provide the behind-the-scenes knowledge of goals' usage, and how these goals are a fundamental part of a complete analysis of a problem domain. A short template for documenting stable analysis patterns was used for the sake of simplicity. By doing so, it will be easier for the reader to cover the provided goals' description in a short time.

CHAPTER 4 Capabilities: The Heart of Business

Capabilities or Business Objects (BOs) are the heart of business. They implement the entities that realize a business concept (Patel, Sutherland & Miller, 1998). Such a responsibility allows them to specify, execute, and track internal norms or comply with logical steps across a spectrum of environments. Like goals, they are intrinsic parts of the Knowledge Maps. Their main responsibility, within the KMs' standpoint, is to interpret a goal's definition, to encapsulate the processing points where this goal is achieved, and to interact with workflow participants. Additionally, capabilities have a major effect on business processing (Lawrence, 1997). They are able to execute the logical steps pertinent to their workflow in parallel, reducing substantially the cycle time necessary to handle a particular goal. Thus, for example, a common task that requires 10 logical steps can have 5 of its logical steps executed in parallel (Lawrence, 1997). Likewise, capabilities support the execution of their logical steps manually or in sequence. Likewise, capabilities support the execution of their logical steps manually or in sequence.

Capabilities represent complete workflows responsible for specifying and managing a stable set of processing tasks. To handle these tasks, these workflows are expressed as sets of smaller chunks of functionality that are easier to understand than high-level complex processing tasks (James, Hawick & Coddington, 2000). Basically, capabilities are designed with a high cohesion of their individual instructions. The definition of these instructions is completed by means of creating a set of services that comply with capabilities' roles and responsibilities (Fayad, Sanchez & Hamza, 2004).

Capabilities complement the goals of a concept. They act as the goals' workhorses. In fact, they are the business workhorses of the goals. These capabilities or workhorses' behavior is driven by a set of enduring business rules, which are encapsulated by goals. Each capability has an ultimate goal. This goal is responsible for constraining the capabilities' internal behavior towards a non-deterministic outcome without causing stragglers. The idea of literally embedding the capability's ultimate goal within the capability's workflow is to ensure a business-centric behavior consistent with the rationale or goals of interest. This idea corroborates two things: a capability without a goal is not a capability, and capabilities are recursively related to the goals without losing generality. Together they form the basis for the generation of many applications across the spectrum of interest, such as Transactions and Services in Service Oriented Architectures (Fayad & Kilaru, 2005, p. 108 – 115).

Similar to goals, capabilities are enduring artifacts, but with a minor difference: they are externally adaptable via extension points or hooks (Fayad, Hamza & Sanchez, 2005). Their adaptable nature can be determined by examining the relationships between the underlying business and its direct application, as well as the application of the right "hooking code." These relationships can be inheritance, aggregation, and associations. The "hooking code" is responsible for weaving business and the actual application together on a real-time basis. Hooks create an environment where capabilities are able to attach to any IO without changing the capabilities' internal structure and without a bit of chance of a collapse.

Further sections introduce how capabilities are part of the workflow metaphor specification and how they consolidate a complete goal processing. Concretely speaking, they make sure you are properly achieving the goal that you discovered during the analysis phase in a complete and accurate manner. That is, they represent how the goal definition should happen, rather than focusing on what should be done: the solution space. The following section represents such a workflow metaphor and its role when facing a set of undetermined contexts.

Workflow Metaphor

Metaphors are useful for introducing a new idea or concept to particular cast (Odell, 1998). Their application to business objects is quite common, especially for specifying and managing their workflows (Fayad, Hamu & Brugali, 2000). In the capabilities' domain, these workflows or business processes are defined with a stable nature in mind. Nonetheless, they provide the right facilities to cope with determined contexts of applicability. Using workflows as a metaphor is interesting, since it allows one to capture and understand those essential "high-level" sequences of activities required to fulfill system behavior. In fact, they dramatically allow complex business processing to be streamlined among workflow participants in determined environments (Fayad, Hamu & Brugali, 2000) and ease their execution and management.

Workflow metaphors have a strong effect on business process definition and understanding. They provide the capabilities for constructing enduring business processes. They also ensure that each of the activities they enclosed is an essential activity, a complete one, of the target environment or subject matter. The activities'

interactions, internal and external, cover detailed (enduring) and dynamic (nested) processing according to the target environment's conditions and opportunities. To assure the proper execution and management of these activities, business objects or capabilities must contain the right services. Using the mantra "divide and conquer" as the starting point, one could write that a proper and focused definition of a capability's services depends on how you distribute the load of each service to achieve the capability's responsibility; hence providing a proper workflow execution.

Workflows of capabilities are expressed as set of smaller chunks of functionality that are easier to understand than "high-level" complex processing tasks. This functionally is one to one mapped with the services the capability provides. How to define these services will determine how accurate the workflow is and how the enduring quality is perceived along the entire workflow. The identification of the services of capabilities' workflows will not be covered in this thesis. However, the process to identify these capabilities will be described. This next sub-section describes such a process.

Dealing with Capabilities: Identification and Assessment

Capabilities are adaptive concerns that represent the solution space of a problem of interest. They embody knowledge skills, processes, and abilities associated with execution of a specified course of actions or actions workflow. Due to their adaptive nature, they ensure a reduced cycle time for coping with a vast number of transient requirements and handling other goals and capabilities in a determined topology of patterns. This peculiarity of capabilities allows both on demand adaptation and flexibility

against IOs or transient aspects, and on demand scalability of the environment to expand the abilities needed to achieve a goal. The above behavior of capabilities is, in the end, translated as a faster return of investment (ROI) while still maintaining a high quality solution. This means that a customer could manage his or her products' functionality (e.g., add, remove, override, extend, etc) when facing new requirements without the necessity of reinventing the wheel.

Developers tend to be more familiar with capabilities than with goals. One reason is they are usually exposed to capabilities on a daily basis. However, this "familiarity with capabilities" does not mean that capabilities are obvious to extract whatsoever; in fact, their extraction is a challenging activity due to different reasons. For instance, these are the common difficulties experienced when trying to extract capabilities:

- 1) Software practitioners don't know if they have discovered a capability or a goal due to the conceptual nature of both. For example, a collection can be seen as both a goal and a capability. Basically, there is a fuzzy line between goals and capabilities' identification.
- 2) Software practitioners have difficulty knowing if they have discovered the right capabilities for a determined goal. For example, what are the capabilities of Branding? By reading a problem statement, practitioners will say: Position, Advertisement, Promotion, Market, Product, Brand, etc. However, they should be looking for the right ones: AnyEntity, AnyBrand, AnyMechanism, AnyParty, and AnyIdentity (Sanchez, 2005).

- 3) Software practitioners misunderstand the nature of capabilities. Instead of thinking in terms of the enduring principles that drive their business, they start thinking in terms of an application's artifacts. From the previous example, one has the following application centric artifacts, which are part of the Marketing field: Product, Market, Promotion, etc. How about if one used branding for identifying humans in the USA? It is obvious that these objects will not be part of the solution.
- 4) Belaboring the rareness of a one-to-one mapping between goals and capabilities (Hamza & Fayad, 2002), a goal can be mapped into several capabilities. This increases the complexity of extracting the right capabilities.
- 5) There are also cases when candidate capabilities are off-content and are thus recursively related to other capabilities and not to goals. This is known by delving into the capability's context of applicability.

In (Hamza & Fayad, 2002), Hamza and Fayad described a process for extracting capabilities that support a determined goal of interest. This process conveys the idea that capabilities must be in context, so that practitioners will not get bogged down with irrelevant details (e.g., application specific details). This process was addressed from a point where one already knew the goal of the subject, but that goal did not have capabilities. In this chapter, the focus will be on capabilities perusal - "how does the subject do it?" Capabilities are examined in two ways. First way is when there is a goal with no capabilities, and second when there is one capability with no goal. Both examination ways are assisted by a set of heuristics or questions.

Before providing the process for identifying capabilities, one needs to understand the two ways for examining capabilities. Firstly, one will have, in most cases, a goal with no capabilities. In this case, the main tasks are to identify those capabilities and their relationships with each other, and to provide a harmony between them and their goals. Secondly, each capability will have a second level of abstraction where it is interacting with other capabilities to satisfy an internal goal. In this case, the main tasks are to identify the internal goal of an examined capability and to find other capabilities interacting with it. You can use the process described in Chapter 3 to identify this goal, but now the subject of interest will be the selected capability.

A Goal with No Capability

Even though capabilities identification seems to be less complicated than goals identification, it is advised for all practitioners to provide the same amount of care and dedication to this process as to the goals' ones. To support this dedication and care, this section provides a set of questions and heuristics that will guide you throughout the entire identification process. These questions and heuristics are listed herein:

- 1) Set the context where all candidate capabilities will be extracted.
Determine the context by asking "what is the subject of interest?"
- 2) Identify the goal(s) of interest by asking "what is the subject for?"
- 3) Determine a focused problem or subject understanding by asking "can one divide this subject into smaller chunks of understanding?" To support this step one uses the following questions:

- a. How can one approach the underlying goal?
 - b. What does one need to fulfill this goal?
 - c. What is one looking for with this goal in the determined subject of interest?
 - d. Who is going to use the goal?
- 4) Filter the entire list of found capabilities.
- a. Does the outcome apply to the goal itself, or is it just part of a wish list of the stakeholder? If it is part of a wish list, this is not a good candidate capability.
 - b. Can the subject matter exist without this capability? Or is the capability redundant within the subject of interest? If so, this is not a good candidate capability.
 - c. Does the outcome have a physical representation in application-specific environments? If not, this is not a good candidate capability.
- 5) Use the prefix “Any” as a generality indicator for both atomic and non atomic capabilities. For instance, the stereotype BO will be used for atomic capabilities, and the “Pattern-BO” for non atomic capabilities. There is an exception for this naming rule. Capabilities that represent types of an element will follow a determined naming convention: The UnitName + Type keyword. For example, types of resources will be renamed as “ResourceType.”

- 6) Evaluate the list of candidate capabilities to assure their accuracy and relevance to the goal of interest. The following questions can be used to support this step.
 - a. What is the relationship between the capability and the target goal? Is it purely obscure or confusing? Or is it clear? If it is clear, this may be a good candidate capability.
 - b. Is the capability's execution and management bound to external conditions and abilities related to a specific context? If so, this is not good candidate capability.
 - c. Does the capability comply with the so-called stateless class definition? If so, this is not a good candidate capability. A good example of this is (Chotin).
- 7) Model the found capabilities and goals using UML notation.

Next subsection concentrates on the second way for dealing with capabilities. It indicates that one must delve into the second level of abstraction of certain capabilities, define their ultimate goals, and define their collaborators.

A Capability with No Goal

Once you have identified all the capabilities of a determined goal, the next step is to take each capability in isolation for immediate perusal. During this perusal, the ultimate goal of the isolated capability will be identified. In addition to this, one identifies, if they exist, the rest of the capabilities that complement the isolated capability's ultimate goal.

Capabilities have their own internal goal to fulfill. The fulfillment of this goal depends on how well software practitioners have examined the isolated capability's structure in search of the pieces that synthesize and realize, as a whole, the rationale of this capability in question.

To ease this examination process, this section will provide a set of questions and heuristics that will guide software practitioners throughout the entire examination process:

- 1) What is the subject of interest? In this case, the selected capability is the subject of interest.
- 2) What is the subject matter for? The process described in Chapter 3 is used to come up with the right goal.
- 3) What are the core elements of the subject matter? This question is addressed by means of following these steps:
 - a. Specify a set of scenarios where the subject matter is present.
 - b. Describe each scenario by identifying the pertinent business issues or rules that drive the subject matter's realization.
 - c. Make sure that each scenario is described from a domain-independent perspective. Table 4 illustrates the scenarios and the business rules that were extracted from them.

Table 4. The Scenario Format

Scenario 1: Scenario Name	
Definition:	Business Rules
Concept of interest' definition	List of business rules of the concept

- d. Abstract the business rules or issues shared by all the scenarios.
- e. Identify the formal axioms that constrain business rules' application across several scenarios.
- f. Does each of one the capabilities have a physical representation? If so, these are tentative core elements of the subject matter.
- g. Discuss the results with the technical cast and look for general consensus of the core elements of the subject.
- h. Continue with this process until you have covered all the possibilities.

In order to support the previous process, examples are provided (in the next sub-section) where all the questions and heuristics previously described.

Capabilities' Identification Process: An Example

To start with this sub-section, a possible goal will be selected as example to explore in detail and find its capabilities. One can walk through the process's steps with the CRC cards example described in (Fayad, Sanchez & Hamza, 2004). The following illustrates the application of heuristics and the questions for extracting the capabilities of a goal.

- 1) Determine the context by asking "what is the subject of interest?" In this case, it will be "CRC cards."
- 2) What are CRC cards for? The result: one of CRC cards' goals is Brainstorming.

- 3) Can one divide the brainstorming concept into smaller chunks of understanding? The result: yes. See below.
 - a. How can one approach the brainstorming concept? The result: *location* where a participant will brainstorm, *rules* for doing brainstorming, *time* for brainstorming, brainstorming *media*, etc.
 - b. What does one need to fulfill this brainstorming goal? The result: one needs *motivation* or *interest* for doing brainstorming, *processes* for doing brainstorming, *forms* of brainstorming, the *target* context where brainstorming is applied, the specific *topic of interest*, *engagement* among participants, etc.
 - c. What is one looking for with the brainstorming goal in the CRC cards context? The result: to complete a brainstorming *session*, to explore a *context*, to produce a list of *new ideas* or *candidate classes* that will be represented by CRC cards, *ideas assessment*, etc.
 - d. Who is going to do brainstorming in the CRC cards context? The result: *facilitators*, *analysts*, *designers*, *scribers*, etc.
- 4) Filter all the identified capabilities (in italics) to avoid getting bogged down with irrelevant details associated with application side of brainstorming.
 - a. Does the outcome (found capabilities) apply to the brainstorming goal itself, or is it just part of a wish list of the stakeholder? The result: *new ideas* and *idea assessment* will be removed.

- b. Can the CRC cards approach exist without some of the found capabilities? Or are some capabilities redundant within the CRC cards approach? The result: the capabilities, such as *time* for brainstorming, *rules* for brainstorming, *location* where brainstorming is being done, and the brainstorming *session* will be removed.
 - c. Does the outcome (found capabilities) have a physical representation? The result: this capability is removed: *motivation* or *interest*.
- 5) Use the aforementioned naming conventions with the left capabilities.
The result: AnyMedia, AnyForm, AnyContext, AnyEngagement, AnyParty, AnyTopic, and AnyMechanism.
- 6) Evaluate this list of capabilities.
 - a. What is the relationship between each one of the capabilities and the brainstorming goal? The result: AnyTopic capability will be removed since its association with the goal is vague and redundant. AnyContext is already covering the AnyTopic role.
 - b. Is each one of the capabilities' execution and management bound to external conditions and abilities related to a specific context? The result: none of the current capabilities are bound to external conditions. Therefore, they won't be removed.

- c. Does each one of the capabilities comply with the so-called stateless class definition? The result: none of the current capabilities are stateless classes.

- 7) The final list of capabilities is: AnyMedia, AnyForm, AnyContext, AnyEngagement, AnyParty, and AnyMechanism.

The second phase is to pick and explore one capability at a time. This is done with the sole purpose of finding each capability's internal goal, along with other capabilities that may complement the picked capability's internal goal. To support this process, the AnyMechanism capability will be used as the subject to be explored.

- 1) What is the subject of interest? In this case, it is the AnyMechanism capability.
- 2) What is AnyMechanism for? The process described in Chapter 3 is used to come up with the right goal. After following this process described in Chapter 3, one comes up with the following goal: *Computing*.
- 3) What are the core elements of AnyMechanism? This question involves the following:
 - a. Specify a set of scenarios where AnyMechanism is present. Use the scenario format illustrated in Table 4.
 - b. Describe the business rules of the AnyMechanism capability.

Table 5 shows the result.

Table 5. The AnyMechanism's Scenarios

Scenario 1: Chemical Reactions	
Definition:	Business Rules
An atomic process that occurs during a chemical reaction.	A step-by-step sequence of steps or reactions that influence a chemical change. It is a process that occurs in natural phenomena. Represents the implementation of an algorithm.
Scenario 2: Earth's Sunlight Process	
Definition:	Business Rules
A process that has occurred on earth due to a chain of causes	A process that occurs on earth according to a determined time period. Certain parameters determine when the light will be perceived on earth. A process that consists of other processes, such as Earth Orbit, Sun, Nuclear reactions, etc.

- c. Take the business rules shared by all the scenarios. For the sake of simplicity, not all the business rules used to extract the atomic structure of AnyMechanism were included. Based on the scenarios, the following capabilities were found: AnyAlgorithm, AnySequence, AnyPeriod, AnySignature, AnyType, AnyClass, and AnyMedia.
 - d. Define the pertinent axioms of the AnyMechanism capability. For example, in order to run the sequence of steps, a period must be defined, etc.
- 4) The final list of capabilities, along with the AnyMechanism goal, is: Computing (goal), AnyMechanism, AnyAlgorithm, AnySequence, AnyPeriod, AnySignature, AnyType, AnyClass, and AnyMedia.

Knowledge Maps Capabilities

This section covers the capabilities of Knowledge Maps. There are a total of 9 capabilities driven the Knowledge Map's formation. The table summarizes these capabilities for a quick reference.

Table 6. The Capabilities of Knowledge Maps

Capability	Description	Provided?
AnyMechanism	AnyMechanism represents the process or logical steps to perform a determined activity.	YES
AnyView	A view of a model that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.	YES
AnyModel	It visualizes the relevant details of a subject or discipline while ignoring the irrelevant details.	YES
AnyLevel	It represents the level of abstraction that a concern or concept can be represented.	NO
AnyContent	This deals with the sum or range of what (patterns) has been perceived, discovered, or learned in a particular discipline.	Yes
AnyContext	This defines what elements are/are not part of the problem under discussion.	NO
AnyArchitecture	This refers to the integration of two or more patterns (Booch, Rumbaugh & Jacobson, 1998).	NO
AnyPattern	This is the best solution for a set of recurring problems or events.	NO
ExtensionPoint	They represent "the slots, knobs, and dials that must adjust in order to adapt the framework to your context" (Booch, Rumbaugh & Jacobson, 1998).	NO

Capability 1

- Name. AnyMechanism Stable Design Pattern.
- Context. The pattern is trying to capture the essentials or core knowledge of any mechanism concept whenever it appears.
- Problem. How to model the core abstractions of a concept that spans multiple application domains.

- Solution and Participants.
 - Solution. Figure 12 illustrates this solution.

Figure 12. The AnyMechanism Stable Design Pattern

- Participants.
 - Classes.
 - AnyMechanism. It represents the process or logical steps to perform a determined activity.
 - Patterns.
 - AnyMedia. Identifies and defines the media upon which the mechanism will be executed.

- AnyClass. Classifies the common properties and behavior for a set of specific kinds of objects.
- AnyType. Names the interfaces that form the mechanisms. It is classified by the provided signatures of a set of services.
- AnySequence. Represents the logical order in which a set of instructions is executed in a determined period.
- AnyPeriod. Indicates the amount of time required by the sequence of steps to be executed in accordance with certain algorithm.
- AnyAlgorithm. Indicates a set of step-by-step rules that produce a determined outcome or solution.
- AnySignature. Represents the specification of an instruction that is part of a mechanism.

Capability 2

- Name. AnyView Stable Design Pattern (Fayad, Islam & Hamza, 2003).
- Context. Represents the core knowledge for the situations in which a view may occur.
- Problem. How does one model the core knowledge of a concept that spans multiple application domains?

- Solution and Participants.

- Solution. Figure 13 illustrates this solution.

Figure 13. The AnyView Stable Design Pattern (Taken from Fayad's patterns archive)

- Participants.

- Classes.

- AnyView. Represents the view of a collection of data (the model).
 - Viewing. Describes the viewing process and rules.

- Patterns.

- AnyParty. Represents the viewer. This viewer perceived the mapped view of a requested model.
 - AnyMedia. Identifies and defines the media upon which the views are mapped and transmitted.
 - AnyProperty. Describes the properties that govern specific kinds of views.

Capability 3

- Name. AnyModel Stable Design Pattern (Fayad, Islam & Hamza, 2003).
- Context. A model is a significant representation of a subject.
- Problem. How does one capture the core knowledge of a model?
- Solution and Participants.
 - Solution. Figure 14 shows this solution.

Figure 14. The AnyModel Stable Design Pattern (Take from Fayad's patterns archive)

- Participants.
 - Classes.
 - AnyModel. Describes a model of a determined application (e.g., plane manufacturing).
 - Modeling. Defines the modeling process.
 - Patterns.
 - AnyParty. Represents the modeler; the one responsible for building the data models.

- AnyMedia. Identifies and defines the media upon which the models are built or exchanged.
- AnyProperty. Describes the properties that govern specific kinds of models.

Capability 4

- Name. AnyContent Stable Design Pattern.
- Context. From an informational perspective, content represents any type of digital information, such as web pages' content, files' content, etc.
- Problem. How does one capture the core knowledge of any content focusing on the information category?
- Solution and Participants.
 - Solution. Figure 15 shows this solution.

Figure 15. The AnyContent Stable Design Pattern (Taken from Fayad's pattern archive)

- Participants.
 - Classes.
 - AnyContent. Describes any type information in a digital context.
 - Patterns.
 - AnyActor. Identifies the content's generator.
 - AnyMedia. Identifies/defines the media upon which the contents are generated and exchanged.
 - AnyForm. Identifies the different types of content in a digital context, such as text, images, and audio.
 - Knowledge. Represents the actors' gained experience.

Last Remarks

By applying the Software Stability Concepts approach to the definition of capabilities, one can conclude that capabilities are true BOs that allow software practitioners to achieve determined goals in a systematic manner. They will comprise a focused logical processing or workflow consistent with a set of enduring business rules.

The following summarizes the information covered in this chapter:

- 1) Capabilities are knowledge-centric business objects with an ultimate goal in mind. A capability without a goal means a capability with an obscure purpose in a determined system.

- 2) Capabilities are internally stable and externally adaptable. This means capabilities development involves both the internal norms and compliance-logical steps forming the capabilities workflow, and the comprising of “hooking code” that enables capabilities to adapt to external application-specific environments.
- 3) Capabilities collaborate with other capabilities towards a common goal, which is the extension and adaptation of their periphery (behavior and structure) to maximize their goal’s success. The focus is on the implementation of hooking points where the capabilities can associate other capabilities, rather than focusing on the capabilities per se.
- 4) Capabilities’ workflow inter-operability definition is defined with a role in mind, which enables the responsibility of each of the capabilities involved in the goal’s resolution. Being aware of the responsibility of each capability will enable a designer to create the right abilities or services of a particular capability.
- 5) Dynamic change of capabilities’ abilities is allowed. Capabilities are embedded with the ability to dynamically change their abilities through extension points which are responsible to add, remove, update, override, and extend functionality on a real time basis.

In summary, the capabilities of a discipline are aspects attempting to encapsulate the business processes and categories of a business-centric theme or goal. When they are directly associated with their goals, they form a synergetic force that is indented to represent the groundwork of any discipline’s understanding: the Knowledge Maps.

CHAPTER 5 Knowledge Maps: A Roadmap for Building Systems

One can think of “knowledge” as the mirror of experience gained by practice and/or study of a particular discipline. Once this knowledge is acquired by an individual, it can be used to help the individual to avoid experienced pitfalls and sloppy processes to occur again. In addition to this, one can create new or streamline previously addressed environments in the search of cost-effective and reusable solutions. So, the question now is how one can represent such knowledge in a straightforward and coherent manner, so that individuals can use it repeatedly to solve recurrent problems.

Existing software patterns and pattern languages emerged as the answer to the aforementioned question. They were created to leverage best practices and lessons learned in the form of standard solutions and architectural styles to solve a set of recurrent problems (Schmidt, Fayad & Johnson, 1996). Software patterns are those standard solutions, while pattern languages are those architectural styles that encompass a set of interrelated patterns (Appleton). Software patterns can be considered as a particular recipe for a meal, while pattern languages are considered as the cookbook containing a set of recipes.

Now that some simple analogies that explain what software patterns and pattern languages are were provided, it will be adequate to start exploring what their shortages are in terms of context capturing, organization, and synthesis. Since their first appearance, about a decade ago, software patterns and pattern languages have been thought of as great techniques for reducing and improving software development lifecycle (Buschmann, 1996; Coplien, 1996; Devedzic, 1999; Fernandez, 1998; Fowler,

1997; Gamma, Helm, Johnson & Vlissides, 1995; Oestereich, 1999; Schmidt, Fayad & Johnson, 1996). In fact, there have been successful stories on their application, especially in the development of large and small scale solutions (Buschmann, 1996; Fernandez, 1998; Fowler, 1997). However, the completion of a particular solution is not always a strong indicator of its efficiency. For instance, even though there have been successful stories of using software patterns and pattern languages, their applications have not been as coherent or straightforward as they should have been.

Most software patterns and patterns languages are created after an exhaustive software development practice – sometimes with good experience, sometimes with poor practices (Gamma, Helm, Johnson & Vlissides, 1995; Laplante & Neill, 2006). Throughout these practices, developers are able to find themselves applying the same fine-grained solutions repeatedly over specific problems. At the end, these fine-grained solutions were turned into patterns that would solve problems of the same sort. However, when it comes to the creation of large-scale solutions out of individual patterns, practitioners realize that is neither an easy nor a systematic task. As a consequence, the reduction and ease of the software development lifecycle via patterns is still a dream difficult to accomplish.

Knowing that the idea of creating and integrating patterns still constitutes a dream has motivated the thesis author to work on a new approach for creating systems of patterns. This approach is known as the Knowledge Maps. This approach facilitates a focused roadmap for knowledge sharing and knowledge reuse. This roadmap will guide software practitioners throughout the process of applying and weaving software patterns to build systems. To better understand this roadmap it will be necessary to describe some

of the properties of the Knowledge Maps, such as knowledge partitioning, knowledge intersection, measurement, reusability, stability, infinite software architecture generation, ROI, remote knowledge, context awareness, and generality. These properties will be described in subsequent sections of this chapter.

Knowledge Maps Representation: Structure, Quality Factors, and Properties

Another way to define Knowledge Maps is as a topology of software patterns. First of all, they are a topology of software patterns because they represent a logical arrangement of patterns according to patterns' rationale and nature. Second of all, the Knowledge Maps, as a topology, facilitate a flexible and coherent environment where practitioners can systematically plan, select, and weave patterns to build systems.

The building of systems is possible by two means. The first one is that part of the creation of Knowledge Maps includes the definition of a roadmap for knowledge representation and reuse. The definition of this knowledge representation will require an understanding of what knowledge to represent in a particular discipline, how to extract and create knowledge, how to represent knowledge, how to implement knowledge chosen schema, how to adapt knowledge when facing new problems, how to analyze knowledge, and, finally, how to exploit knowledge to create accurate solutions (Cercione & McCalla, 1987). The second one is that the Knowledge Maps facilitate a systematic and a straightforward approach to generate, filter, develop, and deploy the pertinent features of a domain via the Software Stability Concepts approach.

In representing the Knowledge Maps as the groundwork for any discipline and also as the answer to problems, such as pattern composition, traceability, etc., the author of this thesis must illustrate how the Knowledge Maps are structured.

Knowledge Maps Structure

Knowledge Maps' structure consists of five major stages that correspond to the Software Stability Models (Mahdy & Fayad, 2002). These stages form the representation scheme that guides knowledge extraction, generation, and synthesis. Basically, the Knowledge Maps show, by means of using stable patterns, the overall processes to solve a set of problems within a determined discipline (e.g., Performance Evaluation, Software Testing). Knowledge Maps will ensure that these processes are easily understood by both technical and non-technical groups. For example, in online bargain systems, concepts such as negotiation, trading, and customer satisfaction are well known by both managers and technical groups. For this reason, these concepts must be included as part of the solution that will be developed for the online bargain system. This is possible by exploring the domain where these activities happened, who is executing them, the media where they are being executed, which mechanism will be used to execute them, as well as the expected result.

Knowledge Maps' stages emphasize the elements pertinent to the software development lifecycle. These elements are analysis concerns, design concerns, knowledge concerns, development concerns, and deployment concerns. Analysis concerns are represented by goals, design concerns by capabilities, knowledge concerns by stable architectural patterns, development concerns by IOs, and deployment concerns

by quality factors. The course of the collaboration between these elements is shown on a priority basis. It starts from the most crucial element for the subject matter to exist, and then it proceeds to the ones with lower priority. Being a low priority element does not mean that this element is not important. What the level of priority indicates is the order of how each element will be handled in a determined solution.

The elements' collaboration is either permanent or temporary. It is permanent when these elements, along with their associations, are part of a conceptual solution (e.g., a stable analysis pattern). This collaboration is temporary when an already defined conceptual solution needs to be extended by adding an ancillary aspect (e.g., goals connected to sub-goals, or capabilities connected to other capabilities). This temporary collaboration happens either at an analysis patterns level, design pattern level, or at an architectural level.

To ensure consistency on how the Knowledge Maps are represented, a set of symbols, as well as their notation, are provided. This will ease Knowledge Maps' use and deployment. To comply with UML notation, each symbol is also represented using UML standards (Oestereich, 1999).

Knowledge Maps Structure's Notation

The first perspective of the Knowledge Maps targets a specific segment of professionals – the non-technical people (e.g., managers). This perspective attempts to give a global idea of a subject being explored using a common vocabulary that can be understood by managers and their technical employees. As a result, requirements misunderstandings regarding the subject may be minimized.

Knowledge Maps' main categories are represented by twenty-sided polygons (icosagon). Each category has its own color to indicate what its addressed problem is. Firstly, analysis concerns or goals, for example, are colored dark blue. Secondly, design concerns or capabilities are colored burgundy. Thirdly, the synergy between goals and capabilities is colored aqua blue. Fourthly, development concerns are colored green. And fifthly, deployment concerns or quality factors are colored dark green. Figure 16 shows this color distribution.

Figure 16. The Knowledge Maps Main Categories

The main categories' elements are represented by hexagons. The nature of these elements is represented by a gray line drawn between a determined element and its category. The nature of these elements is also perceived by how the elements' names are colored. For example, the elements from the goals category have their names colored in dark blue. Figure 17 shows the entire coloring conventions for each one of the elements in the Knowledge Maps. Regarding their naming conventions, they follow a "camel casing" style (e.g., ClassName).

Figure 17. The Elements of Knowledge Maps' Categories

This happens because stable architectural patterns enclose all the goals (at least two), along with their pertinent capabilities, as a whole and independent base that can be reused elsewhere, regardless of the forthcoming requirements.

Knowledge Maps Structure's Relationship with UML

The second perspective, UML perspective (Oestereich, 1999), of the Knowledge Maps targets another specific segment of professionals – the technical people (e.g., analysts, designers, programmers). Managers, with strong technical skills, can also understand this perspective. This perspective may help them guide their developers throughout the software development lifecycle.

The individual elements that form the structure of the Knowledge Maps have the following meaning within the UML's realm:

- Goals. They represent the specification classes that deal with the problem space – the rationale retrospective of a discipline. Goals are represented as classes as in any traditional class diagram. They follow a “camel casing” style as their primary naming convention. Classes of this sort are marked using the two types of stereotypes (see Figure 19 for more details):
 - <<EBT>> indicates that the classes of this sort are atomic enduring business themes.
 - <<Pattern-EBT>> indicates that these classes have a second level of abstraction which encloses other classes and, in some cases, other patterns. Figure 19 shows this behavior.

Figure 19. The UML representation of Goals

- Capabilities. Like goals, capabilities are represented as classes as in any traditional class diagram. They also follow the same naming convention as Goals. Classes of this sort are marked using either the stereotype <<BO>> or <<Pattern-BO>>. Their main purpose is to achieve the goals of a subject matter. See Figure 20 for details.
 - <<BO>> indicates that the classes of this sort are atomic business objects.
 - <<Pattern-BO>> indicates that these classes have a second level of abstraction. Figure 20 shows this behavior.

Figure 20. The UML Representation of Capabilities

- **Development Scenarios.** They enclose the specific application contexts where stable patterns will be applied. These contexts are represented by a set of classes, along with their interrelations and constraints. Classes of this sort are marked with the stereotype <<IO>>. This stereotype conveys their volatile nature (see Chapter 6 for more details).
 - <<IO>> indicates these classes are objects that proportionally change to the occurrence of new requirements.
 - These classes are attached to the core formed by the stable patterns by using classes with a determined hooking code (a.k.a. Hooks). Classes of this sort are marked with the stereotype <<Hook>>.
 - <<Hook>> indicates that these classes are extensions points of the stable patterns included in the Knowledge Maps.
- **Quality Factors.** Quality factors, within Knowledge Maps, are considered as the deployment *goals* of a discipline. Therefore, they follow the same representation, same stereotype, and naming conventions as goals.
- **Associations.** Associations, in UML, are represented by lines drawn between one or more classes. At the end of each association, multiplicities, role names, navigability, and qualified attributes (if necessary) are indicated. Likewise, they need to have a name; this name is usually set in italics. Since the main purpose is not to give a full description of associations' notation, the reader will be recommended to access the following reference (Oestereich, 1999) for more specific details about associations.

- A Three Layer Representation. The Software Stability Models (SSM) (Mahdy & Fayad, 2002) are the visual representation of the Software Stability Concepts approach. They focus on the separation of concerns of a subject: EBT layer (goals), BO layer (capabilities), and IO layer (context). Figure 21 shows what a three-layer representation looks like (Fayad, 2002a, 2002b; Fayad, Ranganath & Pinto, 2003; Fayad & Altman, 2001; Mahdy & Fayad, 2002).

Figure 21. The Three Layer Representation of Knowledge Maps

How the Knowledge Maps are structured and organized depends on the compliance of certain quality factors. Some of these quality factors were extracted from the enterprise software architectures literature (Fayad, Hamu & Brugali, 2000) and apply to the Knowledge Maps' deployment. These quality factors are illustrated herein.

Knowledge Maps Quality Factors

Knowledge Maps can guide individuals to plan and solve a set of recurrent problems within context. In other words, they will provide the ability to focus on the actual problem, to acknowledge the set of constraints, pitfalls, trade-offs, and to obtain tidbits of advice pertinent to the problem, while ignoring irrelevant details. Such qualities are not conceived just because of magic; rather, they are conceived as the product of complying with a set of quality factors. These quality factors are:

- 1) **Stability.** Knowledge Maps must be defined at a knowledge level using the Software Stability Concepts approach' two primary artifacts: EBT and BO. Due to the enduring and reusable nature of both artifacts, they allow the Knowledge Maps to serve as the base for creating stable solutions capable of facing ever-changing problems without any struggle. See Figure 22 (Fayad, 2002a, 2002b; Fayad, Ranganath & Pinto, 2003; Fayad & Altman, 2001; Mahdy & Fayad, 2002).

Figure 22. The Software Stability Concepts' Stratification

- 2) Scalability. This concentrates on the ability of the Knowledge Maps, along with their enclosed elements, to adapt to evolving needs and insights without unnecessary effort (Fayad, Hamza & Sanchez, 2005). This is done by means of either enabling/disabling software patterns to/from the Knowledge Map or extending/reducing their Knowledge Maps' peripheral without any complication or collapse (Fayad, Hamza & Sanchez, 2005).

Figure 23 summarizes what scalability is.

Figure 23. The Achievement of the Scalability Quality

- 3) Traceability by Rationale. This quality concentrates on the ability of the Knowledge Maps' enclosed patterns or elements to be successfully traced back to their original goal or usage rationale (Hamza, 2002; Hamza & Fayad, 2003a) after their implementation.
- 4) Generative Nature. This quality represents both the ability of the Knowledge Maps to reproduce or originate complete solutions based upon a set of intertwined patterns, and the patterns connective rationale: a set of guidelines for selecting patterns and their interconnection.

- 5) Adequacy. Knowledge Maps can satisfy the requirements of their intended and established purposes, along with their consequences (Fayad, Hamu & Brugali, 2000). In other words, the Knowledge Maps offer “goodness of fit,” regardless of the context of application. This quality extends from:
 - a. visualizing and monitoring each element in the Knowledge Maps,
 - b. efficient modeling and documentation means,
 - c. the ability of attaining and representing objective knowledge and the rationale of what is known,
 - d. searching and recognizing capabilities.
- 6) Verification and Validation (V&V) ability. This quality concentrates on providing the means for verification and validation of selected paths (goals and capabilities synergies) and decides whether the selected path is the most suitable selection or not.

Depending on whether these quality factors are fulfilled, Knowledge Maps’ properties will simplify the process of building cost effective systems.

Knowledge Maps Properties

The proper implementation of the aforementioned quality factors in conjunction with a proper knowledge representation will highlight the properties and benefits of the KMs for building systems from patterns. These properties are: partitioning, intersection, measurement, reusability, infinite software architectures generation, faster ROI, direct and remote knowledge, context awareness, and generality.

- 1) Knowledge Maps' Partitioning Property. This property is driven by a common object oriented technology's mantra, "divide and conquer." In other words, a particular domain is partitioned as follow.
 - a. First, a domain of interest will be broken into a set of sub-domains. Instead of dealing with a general domain that contains a dizzy array of information, one will be dealing with fine-grained sets of understanding where each set will be managed with ease. Figure 24 shows the partitioning of the Modeling domain.

Figure 24. The Partitioning of the Modeling Domain.

- b. Second, each concern that forms a particular domain or sub-domain will be allocated in the patterns' topology based on their nature or purpose: EBT, BO, or IO. Figure 25 shows this particular property of the Knowledge Maps using SSM (see Chapters 3 and 4).

- 3) Knowledge Maps' Measurement Property. The number of goals and capabilities enclosed in a patterns' topology drive software measurement. They are used as indicators for measuring the cost, time, and effort that must be allocated to produce a particular software outcome. For example, imagine if a customer approaches you and requests your services to develop a collaborative environment solution. This customer will want to know how long it will take to develop such a solution, how many classes will be necessary to complete it, how many developers will be required, and how much effort will be necessary from them to assist in the development of this solution. These questions can be answered by simply looking at the number of goals and capabilities that will be required to leverage a solution.
- 4) Knowledge Maps' Generality and Reusability Property. This property deals with the degree to which the enclosed software patterns, or a patterns' topology, can be reused elsewhere. This step also provides the necessary logistics to create on-demand stable architectures. Knowledge Maps ensure that all enclosed patterns are in a proper level of abstraction and generality, so that these patterns can be used as the foundations for solving a set of recurrent problems. Figures 27 and 28 show this property using the Branding Stable Analysis Pattern (Sanchez, 2005) in two different application contexts (e.g., Biometrics, Marketing).

Figure 27. Using Branding in Biometric Systems

Figure 28. Using Branding in Marketing

- a. This pattern is the same for the two application contexts due to the pattern's generic nature (Sanchez, 2005). The only thing that changes is the application context, which it's attached to the core by using extension points (Fayad, Hamza & Sanchez, 2005).

- 5) Infinite Stable Software Architectures. Due to the exploitation of useful patterns synergies from available software patterns, one will be able to generate a vast number of stable software architectures. Therefore, the production of software architectures, as well as the workflow selection, will be accelerated. In addition to this, an intuitive process execution to address reusability, scalability, adaptability, etc. will be provided.
- 6) Faster Return of Investment (ROI). Due the stable and reusable nature of the Knowledge Maps, software architectures will be easier and cheaper to construct. There will be a significant reduction in cost and innovation which means a faster ROI.
- 7) Knowledge Maps' Context Awareness and Assessment. Domain understanding and assessment sit side by side in the conceptualization of the KMs. By using the set of heuristics and questions described in previous chapters, software practitioners will ensure that they have identified or collected the right requirements. The proper collection and identification of these requirements will guarantee a unified and accurate patterns' topology, spanning all the building blocks of a particular domain or discipline.

Knowledge Maps versus Traditional Pattern Languages

This section provides the results of a comparative study between existing pattern languages and the KMs. The dimensions from where these two structures were compared are the Knowledge Maps' quality factors and the properties of both subjects.

After an extensive perusal of traditional pattern languages, the following results were generated:

- 1) There are missing indicators and guidelines for assuring a focused nature or behavior for the pattern languages' provided patterns with respect to a selected problem. There is not a method for deciding which patterns to use and add in a particular pattern language. Therefore practitioners are infected with a "keep adding what it seems to satisfy my requirements" inaccurate behavior. At the end, these actions at the end will provide more problems than visible benefits for software practitioners, such as a greater maintenance and/or re-engineering effort.
- 2) There is no identifiable classification of patterns' rationale within a pattern language's structure. This will bring serious problems, like the macho class problem (Fayad, 2003), where all responsibilities are positioned in one class.
- 3) There is a loss of traceability across all the different stages of software development, especially when addressing the deeper levels of patterns' implementation. There is a traceability problem in the instantiation processes of traditional pattern languages, especially since the generated instances cannot be traced back to their original design patterns (Hamza & Fayad, 2003a). The instances were deduced from the design patterns, but the internal structure of design patterns is invisible in traditional pattern languages (Mary Lynn Manns, et al., 2000).

- 4) There is no clear and systematic process to weave patterns together and create software architectures. This is done on ad-hoc basis; therefore, the accuracy of most of the resulted solutions will be questionable, and open to serious contamination and ripple effects (Coplien, 1996).
- 5) There is a loss of generality of pattern languages' patterns. Therefore, the implementation instances of these patterns will be lost within the overall pattern language's implementation (Wu, Hamza & Fayad, 2003). There is no way to pinpoint the pattern instances once the entire pattern language has been implemented (Cerccone & McCalla, 1987; Mary Lynn Manns, et al., 2000).
- 6) There is no distinction between associate and remote knowledge in traditional pattern languages. This statement leads to the following questions "how can one associate two or more pattern languages of a similar discipline? Or how can one intersect one pattern language with one or more pattern languages from a different (remote) discipline?" The answer to these questions is that traditional pattern languages are lacking of a systematic way to do that; usually this is done at ad-hoc (Connelly, Burmeister, MacDonald & Hussey, 2001, pp. 39-49).
- 7) There is a tendency for traditional pattern languages to be developed for here and now, not for the future. Most of the patterns that they enclose have been built based on artifacts or components targeting specific business's processes. One knows that as businesses' policies, mission, etc. change the processes will change as well; therefore, this means that

traditional pattern languages will change. As a result, there will be more effort, cost, and time invested in maintaining existing pattern languages to cope with new businesses' changes.

- 8) Pattern languages don't provide a retrospective of the subject matter's rationale. This shortage has two effects: first, there is a chance for building the wrong pattern language due to its enclosed element were design based on a wrong analysis. Second, since the intent of the pattern language in a particular domain has been obscured by ignoring its original rationale, how the pattern language deals with the problem they addressed is neither straightforward nor easy.

Throughout the previous section you were able to perceive how KMs have provided solutions to the existing problems and shortages of traditional pattern languages. Now, it is time to illustrate how a few architectural patterns that were generated using the Knowledge Maps.

Stable Architectural Patterns Samples

This section describes a couple of stable architectural patterns that were generated using the Knowledge Maps. A short template to document this architectural pattern will be used. In this template, only the essentials of the architectural patterns will be described. All the samples were extracted from previous work on the Software Stability Concepts approach (Fayad, Islam & Hamza, 2003; Yavari & Fayad, 2003).

Architecture 1

The first Stable Architectural Pattern is called Model-View-Mapping (MVM) (Fayad, Islam & Hamza, 2003).

- Name. Stable Model-View-Mapping Architectural Pattern.
- Problem. How to build a high level architecture model that can provide flexible mapping between any abstract model (which could either be a passive model, such as text, and frames, or an active model, such as animating data) to any abstract views and vise versa.
- Solution and Participants.
 - Solution. Figure 29 illustrates this solution.

Figure 29. The Stable Model-View-Mapping Architectural Pattern

- Participants.
 - Classes
 - Applicability. Describes the application and the purpose for which mapping is needed. For instance, in one application, a simple view that extracts part of the data from the original model is needed for the sake of simplicity.
 - Patterns.
 - AnyModel. Describes the models within the application. The model is a representation of the data within the application.
 - AnyView. Represents the view of a collection of data (the model).
 - AnyParty. Represents both the modeler and the viewer. The modeler is responsible for building the data models in the appropriate abstract level. The viewer requests the model and the mapped view of that model.
 - AnyMedia. Identifies and defines the media upon which the models and views are mapped and transmitted. It also represents the media by which the views are to be displayed (e.g., tablets, PC, mobile phones).

- **Mapping.** Defines the mapping rules between the models and their views. It also determines how this mapping will be performed.
- **Searching.** Searches AnyMedia for the requested application, model, or view.
- **AnyApplication.** Represents the application that is requested by AnyParty.

Architecture 2

This stable architectural pattern is called MRI Architectural Pattern (Yavari & Fayad, 2003).

- **Name.** Stable MRI Architectural Pattern.
- **Problem.** Resonance Imaging (MRI) system provides two-dimensional images of soft cells of different organs. Each image shows a horizontal picture of a particular height. Any issue in the organ would be presented in several consecutive pictures. How can one build an accurate, fast, and easy-to-use application for MRI visual analysis out of patterns?

- Solution and Participants.
 - Solution. Figure 30 illustrates this solution.

Figure 30. The Stable MRI Architectural Pattern

- Participants.
 - Classes.
 - Patterns.
 - Recording. This Pattern-EBT indicates the purpose of using a log in a set of application areas.
 - Interpreting. It provides the means for explaining something which is loose or fuzzy or is not obvious.

- Viewing. It indicates the rules for using a set of distinct views of a specific subject (e.g., system model).
- AnyView. Represents the view of a collection of data (a determined model).
- AnyParty. Represents both the recorder and the interpreter. The recorder is responsible for recording the data models in the appropriate abstract level. The interpreter interprets the recorded data.
- AnyMedia. Identifies and defines the media upon which the recorded and interpreted data is transmitted.
- AnyLog. It represents the canonical logging process itself.
- AnyViewingMechanism. Encapsulates the behavior and properties of mechanisms associated with Viewing in one pattern.
- AnyInterpretingMechanism. Encapsulates the behavior and properties of mechanisms associated with Interpreting in one pattern.
- AnyOpinion. It represents the beliefs about something that will be shared among a set of parties.

Last Remarks

Using KMs is a new movement; therefore, few examples have been created under their wings. These examples were part of a comparative study between traditional pattern languages and the KMs. In summary, KMs found differences are.

- Systematic capture processes and full understanding of the domain where a targeted solution must exist. This will provide a unified domain knowledge discovery and understanding approach, spanning all different categories for stable patterns.
- Great reuse and on-demand context adaptation. IOs are attached to the core according to specific needs via these extension points. These qualities will greatly reduce cost of adaptation or upgrades.
- Full traceability capability across the KMs' implementation. This is accomplished by a two way mapping relationship between stable design patterns and their implementation instances. Therefore, traceability, maintainability, and stability become visible assess in software development of software architectures.
- KMs provide unprecedented flexibility by enabling a business to add or remove functionality from its system on a real-time basis. Therefore, there will be a rapid ROI since software architectures are cheaper and faster to construct, as well as to reuse.
- Rapid software architecture definition and application through a systematic workflow pattern process, and an identifiable pattern composition process.

CHAPTER 6 Development Scenarios: Setting the Stage

The thesis so far has taught one how to identify the underlying goals of any discipline, the workhorses to fulfill them, and the synergy between them to generate knowledge concerns or stable architectural patterns. Now, the context wherein these knowledge concerns will be deployed will be discussed.

Before this discussion, the stage needs to be set to define the required concepts for this chapter, so that all are on the same page. To start, the definition of context will be discussed. After an exhaustive search of context definitions, the following definition matches the requirements of this chapter. According to Anind K. Dey (Dey, 2001)

“Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between the user and an application, including the user and the application themselves.”

This definition allows software developers to mentally associate a given application with a set of design scenarios. In fact, it provides a mental construct that defines elements that are or are not part of the problem domain. Think metaphorically about the context as an imaginary fence that encloses relevant information related to your problem and overlook irrelevant details.

From a conceptual point of view, this fence allows one to generate a pervasive core or base, focused and generic enough, which can host a countless set of fine-grained solutions. From a practical point of view, this fence allows one to utilize this core, via extension points or hooks, and specify their new target environment in a cost-effective

manner. Likewise, this fence allows one to be focused on the problem as he/she implements this core, along with its always changing target environments, keeping one away from getting bogged down with irrelevant information – neither promising more for the solution nor requiring less.

Now, you should have a clear idea of the benefits provided by having a focused investigation of the core knowledge of a discipline, along with its “always changing” applications. One must proceed with the illustration on how to implement this core, as well as its target environments, in a way one can obtain the expected quality and consistency.

Just for the record, you should know that most of the guidelines and coding standards that will be illustrated in this chapter are based on the software framework described in Chapter 8. Moreover, you should realize that it is also important to know that the foundations for implementing this framework are provided in this chapter. Therefore, a recursive association exists between Chapter 6 and Chapter 8.

From a development point of view, the implementation of the selected core formed by the EBTs and BOs, along with the target environment formed by the IOs, requires certain clarifications, especially with the terms “classes” and “types,” “hooks specifications,” “user model specifications,” “coding,” “packaging,” and the solution’s “encapsulation” that form a component. The clarifications are provided in the following sections.

Knowledge Maps Implementation Issues

Now, several issues related to the Knowledge Maps' structure and behavior implementation (e.g., EBTs, BOs, and IOs implementation) will be discussed and debated. The goal of this section is to create, among software developers, a more accurate idea of the coding guidelines they will use to implement partial or whole Knowledge Maps (some snippets will be provided). For example, one issue that can be referenced is "Type versus Class" within the Type Oriented Paradigm (TOP) (Fayad, 2003). The analysis will proceed with the impact of using contracts in the Knowledge Map's partial or whole development. Moreover, it will provide the specification of hooks in the KMs. Finally, the use of storytellers, packages, and components as a whole in the implementation of the Knowledge Maps will be described.

Type versus Class within TOP

Within TOP, types are names that denote an interface of an individual (object), and classes are the implementers of the type. In other words, classes encapsulate the implementation of a type (Fayad, 2003). Software developers often overlook and diminish the potential of types, such as the benefits of decoupling a class' specification from its implementation.

The relationship between types and classes is sometimes complex and leads software developers to confusion and antagonism, especially when a class has the same name as its type. To avoid this confusion and antagonism, one must step back a bit and understand, using examples, what types and classes really are in the context of patterns' topologies and in TOP.

Take one of the capabilities of the Knowledge Maps: AnyContent stable design pattern (see Chapter 4). This stable pattern consists of five classes: AnyContent (BO), Knowledge (EBT), AnyActor (BO), AnyForm (BO), and AnyMedia (BO). Each one encapsulates the behavior specification of its respective types. At the same time, some of the classes' behavior specification can be made up of more than one type. In other words, one individual (object) may have one portion of its interface defined by one type, and the portion implemented by another type (Fayad, 2003). One can look at the snippets of the AnyMedia class and the BO type. In this class, two operations defined by the BO type were implemented. Snippet 1 shows the BO type specification.

```
//:~begin
public interface BO {
..... //Some code goes here
void collaborates ( EBT goal );
void connects      ( BO collaborator );
Boolean disconnects      ( BO collaborator );
BO loads            ( String extension );
void disables      ( String      extension      );
.....//Some code goes here
}//:~end
```

Snippet 1. BO Type Specification

Snippet 2 shows the implementation of the BO type.

```
//:~begin
public class AnyMedia extends AbstractBO implements BO{
.....//some code goes here
private Map boRegistry = Collections.synchronizedMap(new HashMap());
public void connects(BO collaborator){
    if(capability!=null){
        .....//some code goes here
        ((AbstractBO)collaborator).determine ( this, collaborator );
        ....//some code goes here
    }
}
.....//some code goes here
private Map ebtRegistry = Collections.synchronizedMap(new HashMap());
public void collaborates(EBT goal){
    ....//some code goes here
    ebt    = goal;
    ....//some code goes here
}
.....//some code goes here
}//:~end
```

Snippet 2. AnyMedia's BO Type Implementation

These two snippets provide enough information to make the point clear about types vs. classes. First of all, within TOP, a class implements a type. A class can implement more than one type. Types can be associated with other types. Types are represented as interfaces in Java. More information can be found at (Bloch, 2001; Oestereich, 1999).

Within the realm of the Knowledge Maps, a template for describing this relationship is provided so software developers can have, at hand, the facility to distinguish between types and classes, and make their software analysis and design more efficient and ordered.

Design Type Specification Template

To ensure an accurate communication among software practitioners (e.g., analysts, designers, architects, and developers) throughout the software design and development process, the right means must be facilitated in order to convey design decisions. These right means are assisted by a determined list of rules to represent designing outcomes, such as

- 1) to ensure that at least 7 operations per class or type within context are available (Fayad, 2003),
- 2) to avoid the utilization of “get and set” methods (Fayad, 2003),
- 3) to ensure that classes and types will follow a “camel casing” naming convention, and
- 4) to ensure that operations or services per class and type are not repeated in other classes and types.

Looking at the following example will make the above rules clear. It uses three classes of the AnyContent stable design pattern (capability). These are AnyMedia (BO), AnyContent (BO), and Knowledge (EBT) within the design type specification template. See Tables 7, 8, and 9.

Design Type Specification Template 1: EBT

Table 7. Type Specification 1

EBT to EBT			
EBT Name	Type Name	Interfaces/Services	Implementation
Knowledge	EBT	release(), ignore(), enlist(), etc.	AbstractEBT
	Knowledge	synthesize(), recognize(), related(), etc.	Knowledge

Design Type Specification Template 2: BO

Table 8. Type Specification 2

BO to BO			
BO Name	Type Name	Interfaces/Services	Implementation
AnyMedia	BO	connects(BO), attaches(IO), collaborates(EBT), etc.	AbstractBO
	AnyMedia	broadcast(), captures(), switch(), etc.	AnyMedia
AnyContent	BO	connects(BO), attaches(IO), etc.	AbstractBO
	AnyContent	add(), remove(), denotes(), scope(), etc.	AnyContent

Table 9. Type Specification 3

BO to IO			
BO Name	Type Name	Interfaces/Services	Implementation
AnyMedia	AnyMedia	broadcast(), captures(), switch(), etc.	Cellphone, Internet, Computer, etc.
AnyContent	AnyContent	add(), remove(), denotes(), scope(), etc.	Text, Diagram, Image, AudioFile, Movie, etc.

In order to ensure that a type's behavior is doing what it has promised to do, a set of restrictions can be applied to the operations' signatures defined by a type. These restrictions are commonly known as "contracts" (Bacvanski & Graff; Bruegge & Dutoit, 2003; Findler & Felleisen, 2000; Lackner, Krall & Puntigam, 2002, pp. 57-76), the overview of which comes in the next section.

The Uses of Contracts in Design Types' Specification

Contracts are useful means to guide the proper use of objects' operations. They ensure that an object's operations comply with the semantics defined by the objects' behavior. Likewise, they establish a set of restrictions or constraints for the operations' signatures defined by a type's interfaces. Constraints or restrictions of this sort include:

- Preconditions. They represent the conditions that need to be fulfilled prior the operation's execution.
- Postconditions. These conditions must be satisfied after any type of operation has ended.
- Invariants. They describe the object's conditions that must be always kept.
- Exceptions. They represent what an object does in exceptional circumstances or when the aforementioned conditions are violated.

In Java, especially when implementing stable patterns, these contracts are written using the `@Pre` directive for preconditions, `@Post` for postconditions, and `@Invariant` for invariants (Bruegge & Dutoit, 2003; Oestereich, 1999). Then, they will be realized within a class's code using a simple "contract" API facility. Until then, some snippets are provided where this facility is used.

The AnyMedia class will be used to show how contracts are written. Snippet 3 shows how contracts are implemented in the AnyMedia class.

```

//:~begin
import core.vnv.contract.*;
/**
 * @Invariant boRegistry.size() >=0
 * @Invariant ebtRegistry.size()>=0
 * @Invariant ultimateGoal !=null
 */
public class AnyMedia extends AbstractBO implements BO{
    .....//some code goes here
    /**
     * @Pre capability != null
     * @Post boRegistry.size() = self@pre.boRegistry.size() + 1
     */
    public void connectsTo(BO capability){
        .....//some code goes here
        Assertion.requires (capability!=null);//pre
        .....//some code goes here
        Assertion.ensure(boRegistry.size()>0)//post
        .....//some code goes here
    }
}
//:~end

```

Snippet 3. Including Contracts in the AnyMedia Class

The next sub-section deals with the description of the Hook Specification Template.

The Hook Specification Template

The Hook Specification Template consists of nine elements. As a whole, these elements provide the blueprint for adapting the core knowledge formed by EBTs and BOs to specific contexts. This template was extracted from the Building Application Frameworks' book (Fayad, Schmidt & Johnson, 1998).

Table 10 illustrates this template using an example from the stable pattern that has been used throughout this chapter – the AnyContent stable design pattern (specifically, the class AnyMedia will be used).

Table 10. The AnyMediaHook's Specification

<ul style="list-style-type: none"> • Name. AnyMediaHook • Requirement. A new media is generated based on a selected Media Type. • Type. 			
Type	Level of Support		
	Option	Supported Pattern	Open-ended
Enabling a Feature			
Disabling a Feature			
Replacing a Feature	✓		
Augmenting a Feature			
Adding a Feature			✓
<ul style="list-style-type: none"> • Area. AnyMedia. • Uses. N/A. • Participants. AnyMedia, Mobile Phone. • Changes. <ul style="list-style-type: none"> new subclass MobilePhone of AnyMedia MobilePhone.broadcast() extends AnyMedia.broadcast() MobilePhone.captures(object) extends AnyMedia.captures(object) repeat as necessary new property MobilePhone.signal where reads signal maps from the AnyMedia.object write signal maps into AnyMedia.object • Constraints. All media used in the AnyMediaHook must be derived from AnyMedia. • Comments. All the derived media that are using this hook must be related to a specific type of media. 			

This template will be used to specify how your core knowledge will be seen in more specific contexts. The next sub-section will provide a description of the AnyContent stable pattern's core from a user's point of view. To do that, User Models or Storytellers are used.

User Model or Storytellers Specification

The user model is just an object diagram that tells a story to a stakeholder. This story indicates all the artifacts involved in the software design, as well as the artifacts interactions. These artifacts include: model name, class names, attributes and operations (sometimes), roles, relationships, contracts, as well as the sequence of events that happens between the included classes (Fayad, 2005). The AnyContent stable design pattern is used to illustrate this user model. See Figure 31 for more details.

Figure 31. The AnyContent Stable Design Pattern's User Model

The remainder sub-sections deal with packaging and composing stable architectural patterns. In order to do that, a brief description must be provided on how to create packages and how to properly turn these packages into components.

Packages Overview

The definition provided in (Oestereich, 1999) states “packages are collections of model elements of arbitrary types which are used to structure the entire model into smaller, clearly visible units.” They represent some sort of a logical and physical boundary for the elements. This boundary defines a unique “namespace” or “home” for a set of model elements, such as classes, types, libraries, subsystems, and other packages.

From a modeling perspective, packages are represented as “folders.” These folders have a name written inside, and sometimes even have stereotypes to written to clarify their purpose. In the case that these folders have other model elements embedded, the package name will be written in its upper left tab. Figure 32 shows as an example of a set of packages.

Figure 32. The Packages Representation and their Relationships in Knowledge Maps

From a coding perspective, using Java, package representation starts with the keyword “package.” Then this keyword is followed by a list of packages (a hierarchy of packages) using their fully qualified names. The above packages are programmatically represented in Snippet 4:

```
//:~begin
package net.hsanchez.km.core.*; //or
package net.hsanchez.km.core.hf.*; //or
package net.hsanchez.km.patterns; //or
package net.hsanchez.km.vnv.*; //and so on

.....//import statements and comments go here
public class .....{
    .....//some code goes here
}//:~end
```

Snippet 4. Packages Representation

Knowledge Maps can be perceived as a large system that contains a set of patterns, architectural patterns, and KMs. To properly manage all these building blocks, each building block must be grouped and organized within its own package. Likewise, some of them can be in the middle of overlapping boundaries. More information about packages can be accessed at (Bruegge & Dutoit, 2003; Oestereich, 1999).

Components Overview

Components are not that different from packages. They also realize some kind of boundary (Oestereich, 1999) for a set of classes, libraries, and other components. They also define the means for structuring and grouping this set of elements as a unit (Oestereich, 1999). The difference between packages and components is that components are executable and interchangeable software units (Oestereich, 1999) with well-defined external interfaces.

There are several environments for components, such as Enterprise Java Beans (EJB), CORBA, etc. Developers have the option to select the environment that meets their needs. For guiding purposes, EJB (Enterprise JavaBeans Technology; Monson-Haefel, 2000) and Java Archive (JAR) (Files, Packaging Programs in JAR) will be used as the selected components' environment to represent KMs. The JAR files will allow one to bundle Knowledge Maps, individual stable patterns and their respective specific contexts, and also stable software architectures, as single executable files. However, for simplicity, only its UML representation will be provided. The picture below will clarify or provide the pertinent representation of components in UML. See Figure 33.

Figure 33. The Components' Representation in UML

Last Remarks

The objectives of this chapter were twofold. Firstly, the chapter would provide the coding recipes for implementing partial or whole Knowledge Maps. Secondly, it would provide their visual representation according to UML standards. These objectives were implemented by providing a set of specification templates, some snippets, and proper examples. More information will be provided in Chapter 8.

CHAPTER 7 Deployment and V & V Concerns: Quality Factors

Deployment and V&V focus on the definition of robust patterns' topologies (KMs) by using a set of quality factors pertinent to a discipline of interest. Creating robust patterns' topologies is a challenge, especially when dealing with very unique disciplines and lack of systematic processes to create them.

Chapters 3, 4, 5, and 6 describe the means for identifying, defining, and evaluating the functional requirements of determined subject, such as the Knowledge Maps subject, etc. Chapter 7 will deal with a different set of systems' requirements, which are the commonly called “-ilities” (Larman) of the system – the non-functional requirements. These non-functional requirements are the systems' operational constraints, which are not related to a system's functional specification (Bruegge & Dutoit, 2003). Within the Knowledge Maps' realm, these non-functional requirements will indicate Knowledge Maps' deployment expectations. Take the deployment or deployment stage as the effective retrospective of the Knowledge Maps' sharing and execution abilities. This retrospective deals with the formulation of a set of goals or quality factors that will drive a robust deployment of a created patterns' topology.

Similar to a discipline, quality factors are embodied as goals, but with a central focus on the intended use of a determined discipline. Using the Knowledge Maps as the discipline of interest, one may encounter quality factors, such as reliability, scalability, traceability, extensibility, usability, and others, none ending with “-ility”, such as performance, accuracy, and so on. The selection of these quality factors will be determined by examining the Knowledge Map as a well-defined and operational unit.

The process for identifying, filtering, and evaluating these quality factors is the same as the one described in Chapter 3. The only thing that software practitioners need to pay attention is the fact that the subject of interest will be a determined Knowledge Map, and not its internal structure, such as rationale. This process can be assisted by means of using heuristics, evaluation/filtering questions, and V&V processes that were used to assess the underlying goals of a discipline. V&V processes will be discussed throughout this chapter.

Deployment of Knowledge Maps

The Knowledge Map's deployment stage represents the set of interrelated activities (e.g., release, installation, and acceptance) that will make Knowledge Maps available and ready to use. These interrelated activities represent, as whole, a systematic process that can be executed and customized according to a determined set of requirements (e.g., expected response time, throughput) defined by stakeholders. To understand this stage, one must tackle it from both a conceptual and practical point of view. The conceptual point of view will give one a clear notion of what terms and semantics are used to understand and realize the Knowledge Maps' deployment. The practical point of view will give one the proof of the concept and benefits of Knowledge Maps' deployment. Together, they will allow one to have the same frame of reference in terms of how to deploy a Knowledge Map.

The Conceptual Point of View of Deployment

This section will begin by defining the requirements that drives the usage of the Knowledge Maps. These requirements are known as non-functional requirements (Bruegge & Dutoit, 2003; Jacobsen, Kristensen & Nowack, 1999; Malan & Bredemeyer). They consist of a set of quality factors and constraints related to the Knowledge Maps' operation. Quality factors, for example, are the properties that stakeholders will care about when using the Knowledge Maps. In other words, they are the factors that will drive stakeholders' satisfaction when using Knowledge Maps. Constraints are the non-negotiable restrictions imposed on Knowledge Map' use. Some examples of qualities and restrictions include: an intuitive GUI's navigational style as a quality, and a prohibited KMs' transformation to text files as a restriction. This quality and constraint will determine the scope of the Knowledge Maps' usage. Therefore, the proper awareness of qualities and restrictions will guide one to properly use the Knowledge Maps.

For the Knowledge Maps, especially since they can be associated with other adjacent maps (within a determined discipline) and with remote maps of a different discipline, their deployment stage involves a proper coordination of a set of determined quality factors. These qualities will change according to each discipline of interest; however, some may commonly appear in all of them.

Some of these qualities were already described in Chapter 5. However, they will be described again in the following table. Table 11 summarizes these quality factors.

Table 11. The Goals of Knowledge Maps' Deployment

Quality Factors	Description
Traceability	In this domain, this quality deals with ability of KMs to be traced back to their original goals after their implementation.
Stability	This quality deals with the ability of KMs to handle future requirements' changes.
Generality	This quality refers to as the ability of KMs to be reused elsewhere and create partial or complete solutions via patterns.
Adequacy	This is the ability of KMs to satisfy the intended and established requirements, along with their consequences (Fayad, Hamu & Brugali, 2000).
Scalability	This concentrates on the ability of KMs, along with their enclose elements, to adapt to evolving needs and insights without unnecessary effort (Fayad, Hamza & Sanchez, 2005).
Verification and Validation	This concentrates on the ability of KMs to be self-verified and validated according to their context of application.

The Practical Point of View of Deployment

Non-functional requirements have the tendency to be conceptualized using loose or fuzzy terms that lead to different types of interpretations (Malan & Bredemeyer). Accordingly, they provide little tidbits of advice or guidance to software engineers, especially when making tough trade-offs decisions or meeting functional goals (Malan & Bredemeyer). The deployment stage of the Knowledge Maps seeks to alleviate, from a practical point of view, this shortage by providing the necessary mechanisms to guide the identification and implementation of non-functional requirements. This practical view represents a combination of best practices, expectations, actual qualities, and promises of what Knowledge Maps can possibly do, such as the qualities described in Table 11, and the Knowledge Maps' properties described in Chapter 5. These elements are then encapsulated in a form of stable software patterns.

The encapsulation and use of these qualities as stable software patterns is considered the proof of the conceptual view of the Knowledge Maps' deployment. Clearly, in order to make deployment activities accessible, complete, and accurate, one must explain or note how the Knowledge Map's context is set and what the purpose of its use is. One way to represent this notion is to represent these activities as quality factors (see Table 11).

As is the case between goals and capabilities, quality factors and their respective capabilities can form insightful synergies. These synergies provide the basis for the generation of an unimaginable number of stable software architectures (see Chapter 5), which in this case, are translated as the realization of specific qualities that stakeholders are after when using Knowledge Maps, such as faster ROI, assembling, and usability.

Table 12 complements the aforementioned deployment goals, illustrated in Table 11, and provide more practical (stakeholder oriented) qualities. Again, the existence of these qualities is bound to each discipline of interest, so these qualities may change from discipline to discipline.

Table 12. Common Stakeholders' Quality Factors

Quality Factor	Description	Potential Capabilities	Provided?
Reliability	In this domain, this quality deals with quality or consistency of Knowledge Maps' services.	AnyService, AnyMetric, AnyMechanism, etc.	NO
Assembling	This quality deals with the ability of KMs to allow systems to be assembled from different components.	AnyEntity, AnyMechanism, etc.	NO
Portability	Represents the ability of KMs to be used in different environments to that for which they were originally created.	AnyEnvironment, AnyMedia, AnyUnit, AnyMechanism, etc.	NO
ROI	The ability of KMs to meet time to market with less cost, time, and effort involved.	AnyLifecycle, AnyMechanism, etc.	NO
Usability	This denotes how easy it will be for any software practitioner to use Knowledge Maps to generate software solutions (non-functional Requirements).	AnyMechanism, AnyContext, AnyActor, etc.	NO

The next sub-section provides an additional set of quality factors that will streamline how KMs are deployed. In addition to this, this sub-section will also indicate what the impacts of properly deployed KMs are on their KMs' generated solutions.

The Quality Factors of Deployment

In Chapter 5 and Table 11, a set of quality factors driving the internal structure and behavior organization/management of the Knowledge Maps was provided. Now, in Table 12, via stable software patterns, an additional set of quality factors dealing with Knowledge Maps' operational process, as a well-defined and self-contained software unit, are described. These quality factors may change or can be modified based on the domain knowledge of the subject at hand, as is the case of the Knowledge Maps. For example, in the domain of CRC cards, these quality factors are Blueprint, Groupware, Adaptability, and Book-keeping. (Fayad, Sanchez & Hamza, 2004), or in the domain of Performance Evaluation, such as Compatibility, Flexibility, Pattern Relatedness, etc. (Fayad & Pradeep, 2004). Now, one needs to be aware of certain activities that are more oriented to the target environment where the Knowledge Maps will be deployed.

The proper implementation of the quality factors illustrated in Tables 11 and 12 is not only concern during deployment of Knowledge Maps. One also needs to be aware of certain activities that deal more with the target environment where Knowledge Maps will be deployed. These activities are part of the software development lifecycle, and they usually include software, Knowledge Maps in this case, configuration, installation, releasing, etc. (Bruegge & Dutoit, 2003). Table 13 will provide these environment-oriented quality factors.

These quality factors attempt to provide the means for assuring that KMs are properly installed, deployed, and used in an efficient way.

Table 13. The Quality Factors of a common Target Environment

Deployment Activity	Description	Provided
Configuration	This is the process that prepares Knowledge Maps for their new target environments. It can be considered the preparation work that developers must perform before installation or release.	NO
Installation	This process is defined the same way as if one was talking about any type of software. It is the initial physical deployment into a determined customer's site. It requires prior configuration.	NO
Releasing	According to (Non-functional Requirements; V&V: Verification and Validation), releasing activity is the interface between development and deployment processes.	NO
Activation	According to (Non-functional Requirements; V&V: Verification and Validation), activation is the activity starting the Knowledge Maps or a Knowledge Maps' tool as an executable unit.	NO
Acceptance	This is the quality that deals with the formal process of accepting a new developed Knowledge Map or a deliverable one.	NO

Deployment – V&V Process

In order to make a Knowledge Maps' solution valid, complete, accessible, operational, and, most importantly, focused with respect to a discipline or subject's specification, one must use the proper verification and validation (V&V) processes during the KMs' entire realization – from goals identification to actual deployment.

In (V&V: Verification and Validation) a definition of verification and validation concepts is provided. In this definition, the target subject is software/system. For the sake of clarity, one can use or imagine that the term Knowledge Maps replace the term software/system. The following are the definitions of V&V (V&V: Verification and Validation):

“Verification is the process of determining if a system meets the conditions set forth at the beginning, or during previous activities of the software development lifecycle, correctly. These conditions are set forth in software requirements, which are usually formally documented.”

“Validation is the process of evaluating the system to determine whether it satisfies the specified requirements and meets customer needs.”

V&V processes are very important during the formulation of KMs. They provide an accuracy assessment of KMs overall structure. In terms of the KMs’ overall quality achievement process, V&V processes are just a single part of it. Basically, to assure quality KMs, V&V processes must be assisted by other aspects found in software engineering, such as configuration management, planning (V&V: Verification and Validation), good analysis, and good design.

From a practical point of view, Knowledge Maps’ V&V processes can be perceived as follows:

Verification processes in Knowledge Maps usually include the following methods:

- 1) Knowledge Map’s Documentation Reviews (after documenting each generated element in the KMs).
- 2) Knowledge Maps’ Inspection (evaluation of past KMs’ conformance).
- 3) Knowledge Maps Auditing.
- 4) Validation of Records (in the design scenarios).
- 5) Knowledge Map’s Simulations (especially in prototyping, and final release of KMs).
- 6) Knowledge Maps’ Demonstration Measurement.

Validation processes in Knowledge Maps usually include the following methods:

- 1) Knowledge Maps Test Case Generation.
- 2) Knowledge Maps Test Suites Generation.
- 3) Knowledge Maps Scenario Testing.

To show a typical testing cycle that can be used in Knowledge Maps (to support the above methods), the following list is provided:

- 1) Goals Identification Phase. Testing should start in this phase to guarantee an accurate solution.
- 2) Capabilities Identification Phase. Found capabilities are evaluated, so that there is an accurate interface between design and implementation.
- 3) Test Planning. Testing strategy and an appropriate approach to evaluate Knowledge Maps' development scenarios are formulated or selected.
- 4) Test Development. Test cases, test suites, test scripts, and scenario testing are generated.
- 5) Test Execution. All the testing elements already defined are executed.
- 6) Test Reporting. Based on the tests execution, each generated result is recorded for future perusal or measurement.
- 7) Continue steps 5 and 6 until necessary.

Obviously, more work has to be done to make the V&V process automatic throughout the Knowledge Maps conceptualization, development, and deployment (e.g., dynamic analysis). For now, this dynamic analysis initiative will be part of KMs' future enhancements.

Last Remarks

In general, the rationale of providing a deployment concern in the Knowledge Maps is to guarantee the proper forming of core sets, along with the enclosed patterns pertinent to the discipline of interest. In short, this stage implies the assuring of Knowledge Maps' quality and authority with respect to the mastering of a particular discipline.

The author has provided in this chapter the respective notions of what the Knowledge Maps' deployment's activities are. The actual documentation, as a pattern, of these activities will be part of future work.

CHAPTER 8 Knowledge Maps Framework: Initial Work

The KMs' framework consists of interfaces, implementations, heuristics, and a hooking facility for context adaptation. The interfaces are the types that the framework supports. These are, EBT, BO, IO, ExtensionPoint, etc. The implementations are the classes that provide a concrete version of these types. Heuristics are the Java coding best-practices that will be used to write effective Java code, such as "prefer interfaces to declare types, avoid prefer static class members over private class members, provide a skeleton implementations of types using abstract classes, etc." (Bloch, 2001). The hooking facility is a set of classes that allows one to adapt specific BOs to any application contexts by means of attaching, associating any type of IO to the BO.

Frameworks' Supported Interfaces

The main interfaces of this framework are: EBT, BO, IO, and ExtensionPoint. There are other interfaces, such as Analysis, Design, and Verifiable, that will be used more with the framework's implementations. For this reason, they will not be described in this chapter. The model representing these main interfaces is illustrated in Figure 34.

Figure 34. The Frameworks' Interfaces or Types

One must get familiar with these types because they are the core of the implementation of KMs' goals, capabilities, development scenarios (as well as their adaptation to new contexts), and deployment goals.

Each type forming the core of this framework, in isolation, is important to the user; however, it is the synergy between them that makes the Knowledge Maps distinct from other approaches (see Chapters 2, 3, 4, 5, and 6). The type EBT is the “what” of the system. It realizes the analysis concerns of a perused problem. This type is associated with a list of classes implementing the type BO, which is the type that represents the “how” of the system. Each BO can be associated with several EBTs; however, there is only one EBT that represents the ultimate goal of the BO. The type IO is a more tangible version of the type BO¹; it represents the concrete application of the BOs. IOs are associated to the BOs by using associations, aggregations or compositions, and inheritance. The ExtensionPoint type handles these forms of associations' generation. The ExtensionPoint type is the one responsible for externally adapting a BO to a set of always-changing requirements (IOs) at an on-demand basis. Examples of extension points are hooks, hotspots, etc.

The Interfaces' Implementations and Types' Structures

In the previous section the interfaces that form the core of the KMs' framework were provided. Now, the basic implementations of these interfaces, abstract implementations, will be provided. This technique is called “Skeleton Implementation”

¹ This is not the generalization-specialization problem; please check Chapter 4 and 6 for more information.

Figure 36. The Framework's Additional Interfaces Implementation

Table 14 shows a summary of the framework's skeleton implementations.

Table 14. The Summary of Interfaces' Implementations

Types or Interfaces	Sub Type	Skeleton Implementation	Concrete Implementation
EBT	Analysis	AbstractEBT	Goal
BO	Design	AbstractBO	Capability
IO	n/a	n/a	As many as necessary
ExtensionPoint	Hook	AbstractExtensionPoint	As many as required

Besides the implementation of this framework's main interfaces, there is a set of supporting classes that complement the main interfaces' responsibilities. These supporting classes, in conjunction with the main interfaces and implementation, provide great benefits to the utilization of the framework. These benefits include the on-demand adaptation of BOs via hooks, the loading of classes (IOs) not necessarily located in the same classpath as the application being developed, the Pre and Post conditions, and the

Table 15. The EBT's Structure

EBT		
Method	Version	Description
introduces(): String	1.0	Gives a name to an EBT
releases(BO): boolean	1.0	Removes a BO from an EBT's structure.
enlist(BO)	1.0	Add BOs to an EBT
divides(EBT)	1.0	Divides an EBT into a set of EBTs or sub-goals
ignores(EBT): boolean	1.0	Ignores a sub-goal from the set of the EBT's sub-goal registry

Table 16. The BO's Structure

BO		
Method	Version	Description
collaborates(EBT)	1.0	The BO is associated with an EBT
connects(BO)	1.0	A BO is connected to another BO.
loads(String): BO	1.0	Loads an Extension Point
disables(String)	1.0	Disables an ExtensionPoint

Table 17. The ExtensionPoint's Structure

ExtensionPoint		
Method	Version	Description
describes(): String	1.0	Gives the name of the ExtensionPoint
binds(BO)	1.0	It binds the ExtensionPoint to a BO.
enables()	1.0	Enables an ExtensionPoint
disables()	1.0	Disables an ExtensionPoint
add(IO)	1.0	Adds new IOs to the ExtensionPoint

Table 18. The IO's Structure

IO		
IO represents a tag that the framework will use to recognize transient aspects or application specific objects.		

Framework's Construction Heuristics

The implementation of this framework was supported by a set of heuristics derived from best-practices in Java development (Bloch, 2001; Forman & Forman, 2004; Simmons, 2004; Stelting & Maassen, 2001), as well as one Java programming's experience. For simplicity, some of the heuristics used in the implementation of this framework were included. The list is provided herein.

- 1) Try to avoid the synchronization of types' interfaces implementation to guarantee good performance. However, there are some exceptions to this rule, such as the ones related to the Singleton design pattern.
- 2) Try to use the concept of optional methods, so that if a type's implementation does not support a particular action it will have to throw an `UnsupportedOperationException`.
- 3) Use interfaces to define types and classes to implement these types.
- 4) Try to use static member classes over the non-static ones (e.g., static nested classes).
- 5) Favor protected constructors over private ones when creating singletons. This will allow you to extend a class and prevent its instantiation from classes of a different sort. This is a good idea for creating extendable singletons.
- 6) When implementing methods, try not to control the checking of parameters validity within your method implementation. Instead, one should try using exceptions to check parameters validity, such as `IllegalArgumentException`, `NullPointerException`, etc.
- 7) Use Java's Reflection API carefully. Even though there are some many benefits of using Java's Reflection API, such as the ability to ignore the source code of components and simply extract their interfaces, etc., the uncontrolled use of it may degrade the performance of your developed solution.

The Framework's Hooking Facility

Once there is a good idea about the framework's main elements (e.g., types and classes), it is important to look at how the framework allows the external adaptation of BOs to cope with requirements changes.

To understand how the BOs are externally adapted to cope with requirements changes, one needs to look at the classes responsible for that adaptation. These classes are: Hook (ExtensionPoint's sub-type), HookStrategy, SubClassStrategy, ConstructStrategy, AssociationStrategy, and AggregationStrategy. Table 19 provides a description of these classes' responsibilities.

Table 19. The Classes of the Hooking Facility

Hooking Facility' Elements		
Class	Version	Description
HookingFacility	1.0	Controls all the calls to the HookingStrategy class.
HookingStrategy	1.0	Represents the parent of all the hooking strategies
ConstructStrategy	1.0	Construct IO classes
SubClassStrategy	1.0	Creates SubClasses of a selected BO
AssociationStrategy	1.0	Creates Associations between IOs and a selected BO
AggregationStrategy	1.0	Creates Aggregations between IOs and a selected BO

A bird's eye view of the entire process for adapting BOs using the hooking facility is provided in Figure 38. Figure 38 shows how a hook is attached to the BO of interest. If a user of the BO decides to adapt this BO to new contexts, he/she has three hook options to choose: extend the BO and specialize the BO's functionality, associate any IO to the BO of interest, aggregate any IO to the BO of interest. The ConstructStrategy class is used only for creating "dummy" classes (IOs without implementation) that will be later implemented by a developer.

Figure 38. The Hooking Facility's Process Overview

For example, to adapt a BO using the SubClassStrategy the user needs to pass as inputs the BO to be adapted, and the output name of the new class. Then, the hooking facility will take this BO and extract, using the Reflection API, the BO's constructors, fields, and methods. Snippet 5 will show the Hook's method responsible for creating the BO's identity subclass.

```
public void adapt(BO cls, String output) {
    Class newBO = UClassLoader.search ("net.hsanchez.km.repository."
                                      + output) ;
    setFirstInputObject( cls.getClass () ) ;
    setOutputClassName ( output ) ;
    setPackageName      ( "net.hsanchez.km.repository" ) ;
    Class nBO           = new SubClassStrategy().createClass ( this ) ;
    if(nBO == null)
        throw new HookingException("Unable to adapt the given BO");
}
```

Snippet 5. Creating a BO Sub-class

The use of the UClassLoader.search (...) method allows the loading of constantly modified classes into memory. It loads and compiles recently modified hooks, so that they are used by BOs at runtime. Snippet 6 shows a code sample of the UClassLoader.

```
....
//why? because we can modify the class and reload it again
//without restarting the application
ClassLoader cl = new URLClassLoader( urls );
cls = cl.loadClass(name );
.....
```

Snippet 6. Dynamically Loading Modified Classes

To adapt a BO using the AssociationStrategy, the user needs to provide the BO to be adapted, the IO to be associated with the BO, the association's multiplicity, and the name of the hook to be generated as inputs. The process is quite similar to the previous strategy. The only difference is that the hooking facility will now extract the fields and methods of the provided IO. Additionally, the hooking facility will generate the appropriate method to handle the association between selected BO and IO. Snippet 7 shows a partial implementation of the method that uses the AssociationStrategy.

```
...
    this.setPackageName ( "net.hsanchez.km.repository" ) ;
    this.setMultiplicity ( multiplicity ) ;
    Class nBO = new AssociationStrategy().createClass ( this ) ;
    if ( nBO != null ) {
        hook = (BO)instantiates ( nBO ) ;
    }
...
```

Snippet 7. Associating IOs and BOs

The third strategy is the AggregationStrategy. This strategy is similar to the AssociationStrategy. The only difference is that the hooking facility will implement the appropriate means for aggregating a list of IOs into a generated hook. Snippet 8 shows partial implementation of the method that uses the AggregationStrategy.

```
...
    this.setPackageName ( "repository" ) ;
    this.setMultiplicity ( multiplicity ) ;
    Class nBO = new AggregationStrategy().createClass ( this ) ;
    if ( nBO != null ) {
        hook = (BO)instantiates ( nBO ) ;
    }
...
```

Snippet 8. Aggregating an IO to a BO

The following sub-section provides a typical usage scenario of the hooking facility. This scenario uses the AnyLog BO and the Receipt IO as inputs. The goal is to create an association between these two classes. Snippet 9 and 10 provide the input code, and the resulted Java file.

An Example: Creating Associations between BOs and IOs

The following snippets illustrate how BOs and IOs are associated using the hooking facility. For the sake of simplicity, a rudimentary example will be given.

Snippet 9 shows the input code that will be used to support this example.

```
....
Hook hey = new Hook();
AnyLog log = new AnyLog();
Receipt receipt = new Receipt();
hey.associate(log, receipt, "AnyLog" );
....
```

Snippet 9. The Input Code

The Snippet 10 will show the resulted java file that was generated given the above input code (see Snippet 9). The generated java file is a hook that can be used within the AnyLog (BO) of interest to streamline its functionality using any behavior from the Receipt (IO).

```
public class AnyLogHook
    extends AnyLog
    implements Hook{
//===== F I E L D S =====
private Receipt _receipt;

//===== C O N S T R U C T O R S =====
public AnyLogHook()
{
    super();
    _receipt = new Receipt();
}
//===== M E T H O D S =====
public Receipt getReceipt(){
    return _chair;
}
public void setReceipt(final Receipt ipCls) {
    if(ipCls == null){
        throw new IllegalArgumentException ("Argument must not be null.");
        _receipt = ipCls;
    }
}
//===== N E S T E D   C L A S S E S =====
}
```

Snippet 10. The Resulting Hook

Last Remarks

This chapter's objectives were to provide a concise view of the KMs' framework and to show, via examples, the hooking facility structure. This first version of this framework tries to facilitate a platform for developing KMs' concerns (stable patterns) in a systematic manner. The benefits of having a built-in hooking facility for the BOs to use are numerous. For instance, developers will reduce their effort in adapting their software to new contexts; they will simply use the hooking facility and adapt their BOs the way they wanted by means of extensions, associations, and aggregations hooking strategies. Another benefit is the one of faster ROI; developers will be able to accelerate their coding deliverables, etc.

CHAPTER 9 Conclusions and Future Work

The work accomplished in this thesis fulfills the requirements that were set up in the beginning of this venture. These requirements were represented by three main questions: how can one classify, develop, and utilize analysis and design patterns together towards a problem resolution? What is the “behind-the-scenes language” that guides the sewing of patterns as a whole? How can one overcome challenges, other than patterns composition problems (patterns traceability, etc.) that can hinder the development of a system of patterns? Throughout this thesis, the answers to these questions were provided. The answers’ backbone was the integration of pattern language techniques and the Software Stability Concepts approach (Appleton; Coplien, 1996; Fayad, 2002a, 2002b; Fayad & Altman, 2001; Schmidt, Fayad & Johnson, 1996).

Is important to write that throughout this thesis, commonly found drawbacks and shortages of current software pattern approaches (e.g., patterns composition, traceability, and generality) were analyzed with the sole purpose of providing a clear direction for this work. As a result of this direction, a standard way for conceiving, building, and deploying systems was provided. This standard way is known as Knowledge Maps or a topology of software patterns. Knowledge Maps are the roadmap for delving into the rationale, business rules, and application contexts of a set of problem domains. This roadmap will provide the means for developing quality software systems.

It was emphasized throughout this thesis that Knowledge Maps’ essence was twofold: a methodology and a visual representation. As a methodology, a set of guidelines, heuristics, and quality factors were provided. These guidelines, heuristics,

and quality factors will ease the process of creating Knowledge Maps. As a visual representation, a set of visual gadgets or symbols that convey how the Knowledge Maps and their enclosed elements look like, and how these interact with other elements or other Knowledge Maps, was provided. In conjunction, both methodology and visual representation serve as the roadmap for building cost-effective systems from software patterns. In addition to this, this roadmap allows management and technical personnel to be synchronized when creating systems based on goals and capabilities.

Knowledge Maps are also useful for improving team dynamics between management and technical personnel. Knowledge Maps do that by creating an environment where the initial clashing of ideas between these professionals, due to their own beliefs and experience, is immediately brought to the table, so that they can be addressed. As a result, these professionals can focus on what the problem really is and not on irrelevant details. This will make the process of creating software solutions fun and interesting, since these professionals are able to broaden the scope and horizon of their discussions. Also, since the nature of some of the KMs' processes is based on trial and error, these professionals will lose their coyness in coming up with ideas, since it became clear that no idea was bad, but only part of the process of getting it right.

The proof of concept of Knowledge Maps was accomplished by providing a framework for developing partial or whole Knowledge Maps. Some of this framework's source code was provided in Appendix B. The completion of this framework partially fulfilled the objective in terms of Knowledge Maps' provisioning and usage. The next section describes the author's future work related to Knowledge Maps.

Future Work

Putting in place Knowledge Maps as the means for creating cost-effective software solutions will not only enhance the levels of communication and interaction between software teams (e.g., managers, programmers), but also it will make the process of creating software fun and interesting. These benefits, however, not only apply to the software realm, but also to other areas of research and development, such as education, management, etc. Therefore, the following list represents only an initial draft of the work that can be done to enhance Knowledge Maps' usage:

- 1) IDE for Knowledge Maps management. There are several projects that can streamline the utilization of the Knowledge Maps in software development, such as the development of an IDE for Knowledge Maps management, the standardization and formalization of the process of building Knowledge Maps or topology of patterns, the development of facilities to automate the process of testing remote knowledge (remote Knowledge Maps), and the generation of software architectures. The completion and combination of these projects will give one additional means to create cost-effective software systems. The additional means include: the automation of software product's dynamic analysis to reduce software development lifecycle, automation of intuitive V&V techniques to assure software quality, and the automation of the process of integrating numerous Knowledge Maps.

- 2) Proper utilization of existing technologies. To create a complete IDE for Knowledge Maps management, more technologies such as RDF, JDOM, XML schema (JDOM; Resource Description Framework (RDF); XML Schema), etc. can be added to one's current work. The addition of these technologies to one's current work will streamline the provided Knowledge Map's framework. These technologies can be used, for example, to make Knowledge Maps portable, reusable and representative to the information they conveyed regardless the environment of deployment.
- 3) Standardize and formalize Knowledge Maps' building process. Another possible project will be the standardization and formalization of the process of building Knowledge Maps using formal languages, such as Object Z, and Z++ (Object-Z; Z++ Language Syntax Chart). The utilization of formal languages will endow Knowledge Maps process with the suffice validity, integrity, efficiency and authority to be properly used by different software practitioners (with different levels of expertise).
- 4) Automation of remote knowledge's testing and software architectures' generation. To automate the testing of remote knowledge and the generation of software architectures, more dynamic methods for information gathering and pattern identification/definition, intuitive verification and validation techniques, and interaction with backend databases can be integrated in one's Knowledge Map's framework. The utilization of these methods will endow the process of building

Knowledge Maps the means for dynamic analysis, integration with other Knowledge Maps, software development lifecycle reduction, and a faster return of investment (ROI).

Last Remarks

The work presented in this thesis brings significant contributions to the software engineering field, especially in the fields related to software patterns. These contributions range from software pattern identification, evaluation, and definition heuristics, Knowledge Maps' formation, evaluation, management and deployment processes, to the initial framework that will assist the development of software using the Software Stability Concepts approach (Fayad, 2002a, 2000b; Fayad & Altman, 2001).

The other aspect of this work was the author's contributions to Knowledge Maps' users. The author's contributions are: the initial framework for developing Knowledge Maps, especially the part that deals with the implementation and testing of EBTs, BOs, IOs, and hooks; the set of stable analysis and design patterns that creates the core of the Knowledge Maps approach. The documentation of these patterns, as well as the source code of this framework is included in this thesis.

In general, the main benefit of the author's approach is to bring to the software practitioners the means for answering three important questions: how can one classify, develop, and utilize analysis and design patterns together towards a problem resolution? What is the "behind-the-scenes" language that guides the sewing of patterns as a whole? How can one overcome challenges, other than patterns composition problems (patterns traceability, etc.) that can hinder the development of a system of patterns? The

combination of traditional pattern languages techniques and the Software Stability Concepts approach gave birth to Knowledge Maps, and hence, the answers to the aforementioned questions.

As perspective, there are plans to develop and enhance the Knowledge Maps approach by means of implementing tools that will ease the process of building, evaluating, and deploying stable software patterns.

REFERENCES

- Abbot, J. (1983). Program Design by Informal English Descriptions. *Communications of the ACM*, 26 (11), 882-894.
- Anton, A. I. (1996). Goal-based requirements analysis. In *Proceedings of the IEEE International Conference on Requirements Engineering (ICRE '96)*, Colorado Springs, USA.
- Anton, A. I., & Potts, C. (1998). The Use of Goals to Surface Requirements for Evolving Systems. *The 20th International Conference on Software Engineering*. Kyoto, Japan.
- Appleton, B. (n.d.). *Patterns, and Software: Essential Concepts and Terminology*. Retrieved from <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html>
- Bacvanski, V., & Graff, P. (n.d.). *Lecture Notes in Design by Contract in Java*. Retrieved from <http://www.inferdata.com>
- Berard, E. V. (n.d.). *Abstraction, Encapsulation, and Information Hiding*. Retrieved from The Object Agency Web site: <http://www.toa.com/pub/abstraction.txt>
- Bloch, J. (2001). *Effective Java: Programming Language Guide* (1st ed.). Wesley.
- Booch, G., Rumbaugh, J., & Jacobson, I. (1998). *The Unified Modeling Language User Guide* (1st ed.). Wesley.
- Bruegge, B., & Dutoit, A. H. (2003). *Object Oriented Software Engineering: Using UML, Patterns and Java* (2nd ed.). Prentice Hall.
- Buschmann, F. (1996). *Pattern-oriented Software Architectures: A System of Patterns*. New York: Wiley.
- Cercone, N., & McCalla, G. (1987). *The Knowledge Frontier: Essays in the Representation of Knowledge* [Symbolic Computation / Artificial Intelligence] (1st ed.). Springer.
- Chen, Y., Hamza, H.S., & Fayad, M.E. (2005). A Framework for Developing Design Models with Analysis and Design Patterns. *The 2005 IEEE International Conference on Information Reuse and Integration, Las Vegas, NV*, 592 – 596.
- Chotin, M. (n.d.). *RemoteObject Stateful vs. Stateless Classes*. Retrieved from http://weblogs.macromedia.com/mchotin/archives/2004/08/remoteobject_st.cfm

- Cline, M., & Girou, M. (2000). Enduring Business Themes. *Communications of the ACM*, 43 (5), 101-106.
- Connelly, S., Burmeister, J., MacDonald, A., & Hussey, A. (2001). Extending and Evaluating a Pattern Language for Safety-critical User Interfaces. *Conferences in Research and Practice in Information Technology Series: Sixth Australian workshop on Safety critical systems and software*, 3, 39-49.
- Coplien, J. (1996). *Software Patterns*. SIGS, New York.
- Devedzic, V. (1999). Ontologies: Borrowing From Software Patterns. *Intelligence*, 10 (3).
- Dey, A. K. (2001). Understanding and Using Context. *Personal and Ubiquitous Computing (2001)*, 5 (4-7).
- Enterprise JavaBeans Technology*. (n.d.). Retrieved from <http://java.sun.com/products/ejb/>
- Fayad, M. E., Hamza, H. S., & Sanchez, H. A. (2005). Towards Scalable and Adaptable Software Architectures. *The IEEE International Conference on Information Reuse and Integration, Las Vegas, NV*.
- Fayad, M., Schmidt, D. C., & Johnson, R. (1998). *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley.
- Fayad, M.E. (2002a). Accomplishing Software Stability. *Communications of the ACM*, 45 (1).
- Fayad, M.E. (2002b). How to Deal with Software Stability. *Communications of ACM*, 45 (4).
- Fayad, M.E, Ranganath, A., & Pinto, M. (2003). Towards Software Stability Engineering. Sixth International Conference on the Unified Modeling Language (UML 2003), Workshop on Stable Analysis Patterns: A True Problem Understanding with UML, workshop #8, San Francisco, CA.
- Fayad, M.E. (2003). Lecture notes in Advance Object Oriented Analysis and Design: Type Oriented Paradigm. Fall 2003. Retrieved from <http://www.engr.sjsu.edu/~fayad>
- Fayad, M.E. (2005). *Lecture notes in Software Architectures. Spring 2005*. Retrieved from <http://www.engr.sjsu.edu/~fayad>

- Fayad, M.E., & Altman, A. (2001). Introduction to Software Stability. *Communications of the ACM*, 44 (9).
- Fayad, M.E., & Kilaru, H. (2005). AnyInformationHiding: A Stable Design Pattern. *The 2005 IEEE International Conference on Information Reuse and Integration, Las Vegas, NV*, 108 – 115.
- Fayad, M.E., & Pradeep, R.S. (2004). A Pattern Language for Performance Evaluation. *The 4th Latin American Conference on Pattern Languages of Programming*.
- Fayad, M.E., & Telu, S. (2005). The Learning Stable Analysis Pattern. *IEEE International Conference on Information Reuse and Integration, Las Vegas, NV*.
- Fayad, M.E., Hamu, D., & Brugali, D. (2000). Enterprise Frameworks Characteristics, Criteria, and Challenges. *Communications of the ACM*, 43 (10).
- Fayad, M.E., Islam, N., & Hamza, H. (2003). The Stable Model-View-Mapping (MVM) Architectural Pattern. Workshop on Reference Architectures and Patterns for Pervasive Computing, The 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03), Anaheim, CA.
- Fayad, M.E., Sanchez, H., & Hamza, H. (2004). A Pattern Language for CRC cards. *The 11th Conference on Pattern Language of Programs, Illinois, USA*.
- Fernandez, E. B. (1998). Building Systems using Analysis Patterns. *Proceedings of the 3rd International workshop on Software Architecture*.
- Files, Packaging Programs in JAR*. (n.d.). Retrieved from <http://java.sun.com/docs/books/tutorial/jar/>
- Fincher, S. (1999). What is a Pattern Language? Chi'99, ACM SIGCHI Conference on Human Factors in Computing Systems in Pittsburgh, Pennsylvania, USA.
- Findler, R. B., & Felleisen, M. (2000). Behavioral Interface Contracts for Java. *Technical Report CS TR00-366, Department of Computer Science, Rice University, Houston*.
- Forman, I. R., & Forman, N. (2004). *Java Reflection in Action*. Manning Publications.
- Fowler, M. (1997). Analysis patterns: Reusable Object Models. Wesley.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Wesley.

- Hamza, H. (2002). A Foundation for Building Stable Analysis Patterns. *Master's Thesis Report, University of Nebraska, Lincoln, USA.*
- Hamza, H., & Fayad, M. (2003). Engineering and Reusing Stable Atomic Knowledge (SAK) Patterns. *The IEEE International Conference on Information Reuse and Integration, Las Vegas, NV, USA.*
- Hamza, H., & Fayad, M. (2004). Applying Analysis Patterns through Analogy: Problems and Solutions. *Journal of Object Technology*, 3 (4), 197-208.
- Hamza, H., & Fayad, M. E. (2002). A Pattern Language for Building Stable Analysis Patterns. *The 9th Pattern Languages of Programs Conference (PLoP'02), Monticello, IL, USA.*
- Jacobsen, E. E., Kristensen, B. B., & Nowack, P. (1999). Characterizing Architecture as Abstractions over the Software Domain. *Position paper for the 1st Working IFIP Conference on Software Architecture (WICSA1), San Antonio, Texas.*
- James, H.A., Hawick, K.A., & Coddington, P.D. (2000). An Environment for Workflow Applications on Wide Area Distributed Systems. *Technical Report DHPC-091, Distributed and HighPerformance Computing Group, Department of Computer Science, The University of Adelaide.*
- JDOM. (n.d.). Retrieved from <http://www.jdom.org/index.html>
- Khadpe, P. (2005). Pattern Language for Data Mining. *Master's Thesis Report, San Jose State University.*
- Lackner, M., Krall, A., & Puntigam, F. (2002). Supporting Design by Contract in Java. *Journal of Object Technology. Special Issue: TOOLS USA 2002 Proceedings*, 1 (3), 57-76.
- Laplante, P. A., & Neill, C. J. (2006). *Antipatterns: Identification, Refactoring, And Management*. CRC Press.
- Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process* (2nd ed.). Prentice Hall.
- Lawrence, P. (1997). *Workflow Handbook*. John Wiley & Sons Ltd.
- Mahdy, A., & Fayad, M. E. (2002). A Software Stability Model Pattern. *Proceedings of the 9th Conference on Pattern Language of Programs (PLoP), USA.*

- Malan, R., & Bredemeyer, D. Defining Non-functional Requirements. *Technical Report at Bredemeyer Consulting*. Retrieved from <http://www.bredemeyer.com>
- Mary Lynn Manns, et al. (2000). Introducing Patterns (or any new idea) into Organizations. Addendum to the 2000 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications.
- Monson-Haefel, R. (2000). *Enterprise Javabeans* (2nd ed.). O'reilly.
- Oestereich, B. (1999). *Developing Software with UML: Object Oriented Analysis and Design in Practice* (1st ed.). Wesley Professional.
- Patel, D., Sutherland, J., & Miller, J. (1998). *Business Object Design and Implementation II*. Springer.
- Non-functional Requirements*. (n.d.). Retrieved from <http://en.wikipedia.org/wiki/ilities>
- Software Deployment*. (n.d.). Retrieved from http://en.wikipedia.org/wiki/Software_deployment
- Simmons, R. (2004). *Hardcore Java* (1st ed.). O'Reilly Media.
- Stelting, S. A., & Maassen, O. (2001). *Applied Java Patterns* (1st ed.). Prentice Hall.
- NLG Project Planning: A Tutorial*. (n.d.). Retrieved from http://e-services.imls.gov/project_planning/
- Object-Z*. (n.d.). Retrieved from <http://www.itee.uq.edu.au/~smith/objectz.html>
- Odell, J. (1998). *Designing Agents: Using Life as a Metaphor*. Retrieved from Complex Adaptive Systems Web site: <http://www.distributedcomputing.com>
- Portability*. (n.d.). Retrieved from <http://en.wikipedia.org/wiki/Portability>
- Reamy, T. (2001, August 9th). Knowledge Maps: An Intellectual Infrastructure for KM. *InsideKnowledge*, 5 (1). Retrieved from <http://www.kmmagazine.com/xq/asp/sid.0/articleid.E3E43AE2-5484-4A4B-9D51-8427D18B6FDC/qx/display.htm>
- Resource Description Framework (RDF)*. (n.d.). Retrieved from <http://www.w3.org/RDF/>
- Salinger, N. A. (n.d.). *The Structure of Pattern Languages*. Retrieved from <http://www.math.utsa.edu/sphere/salinger/StructurePattern.html>

- Sanchez, H. A. (2005). Laying the Foundations for Branding as a Stable Analysis Pattern. *The 19th European Conference on Object-Oriented Programming*.
- Sanchez, H.A., Lai, B., & Fayad, M. E. (2003). The Sampling Analysis Pattern. *Workshop on Timeless and Stable Architectures, IRI'03 Conference, Las Vegas, NV*.
- Schmidt, D. C., Fayad, M., & Johnson, R. E. (1996). Software Patterns. *Communications of the ACM*, 39 (10).
- Telu, S. (2005). Mastering the e-Learning Through Stable Patterns. *Master's Project Report, San Jose State University*.
- V&V: *Verification and Validation*. (n.d.). Retrieved from http://en.wikipedia.org/wiki/V%26V:_Verification_and_Validation
- van Lamsweerde, A. (2001). Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE)*.
- Hamza, H., & Fayad, M. (2003a). On the Traceability of Analysis Patterns. *ChiliPloP 03', Carefree, Arizona, USA*.
- Hamza, H., & Fayad, M. (2003b). Towards a Pattern Language for Developing Stable Software Patterns. *The 10th Conference on Pattern Languages of Programs*.
- Wu, S., Hamza, H., & Fayad, M. (2003, March). Implementing Pattern Languages Using Stability Concepts. *ChiliPloP 03', Carefree, Arizona, USA*.
- XML Schema*. (n.d.). Retrieved from <http://www.w3.org/XML/Schema>
- Yavari, E., & Fayad, M.E. (2003). A Stable Software Model for MRI Visual Analyzer. *The 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003) Poster, Anaheim, CA*.
- Z++ Language Syntax Chart*. (n.d.). Retrieved from <http://www.zhmicro.com/Syntax.pdf>
- Zack, M. H. (1999). *Competing Knowledge* (pp. 81-88). New York: Faulkner & Gray.

APPENDICES

APPENDIX A. Detailed Templates

Knowledge Stable Analysis Pattern Detailed Template

- Name. Knowledge Stable Analysis Pattern
- Known as (Optional). N/A
- Context. Knowledge can be gained through experience or studies. It represents a collection of facts, rules, tips, or lessons learned with respect to anything that must be synthesized to create knowledge. This pattern will be used to represent knowledge synthesis and acquisition.

For example in autonomic computing, knowledge is a collection of information that is acquired through examinations of log files, as well as other types of files, located in local or remote repositories (e.g., Servers/PCs in local networks or in wide area networks). Another example of knowledge application is encountered in the area of customer relationship management (CRM) systems. In these CRM systems, knowledge is collected to understand customers' buying behavior, so that better services are provided, and sales increments are achieved. Knowledge is obtained by means of recording the customers' navigational behavior (clicks) during their shopping session.

- **Problem.** There is a tendency to think of knowledge as individual facts, or any relevant and specific information that a system collects or requires. This perception makes knowledge's implementation different every time it is used, especially since each system may handle a different subject that consist of different information and constrains. Why does one need to re-invent the wheel every time one is dealing with different types of knowledge? This is significant problem that must be overcome. Therefore, one needs to answer the question: how does one encapsulate the main component of knowledge as an aspect, regardless of its context of applicability, and avoid re-inventing the wheel?
- **Challenges and Constraints.**
 - **Challenges.**
 - Encapsulate the default structure and behavior that will be shared among different knowledge implementations.
 - Allow a canonical handling of different set of subjects that may be selected during knowledge acquisition.
 - Robustly cope with a set of distinct actors per application on an on-demand basis.
 - Synchronize the actors' provided skills with the capabilities provided by different mechanisms during the knowledge acquisition activity.
 - Actors must take respective actions according their abilities in varied and uncertain situations.

- Constraints.
 - The domain of interested must be selected before any other aspect or subject selection.
 - Subject's definition must wrap and delegate the default and public capabilities of the mechanism aspect.
 - Before utilizing a specific actor, the actors' skills must be assessed with respect to the selected subject.
 - Knowledge acquisition process is a synchronized process.
 - There must be at least one actor understanding this knowledge.
 - Knowledge can be shared or belong to multiple domains.
 - A single domain can contain at least one subject of interest.
 - A subject can be either atomic (does not contain sub-subjects) or composed (contains at least one sub-subject).
 - A subject involves a set of mechanisms. At least one mechanism needs to be associated with a subject of interest.
 - One or more mechanisms will streamline one or more skills of a determined actor.

- Solution. The solution is divided into the pattern's structure and its participants.
 - Structure. The structure of this pattern is illustrated via a class diagram. Figure 39 illustrates this class diagram.

Figure 39. The Structure of the Knowledge Stable Analysis Pattern

- Participants. The pattern's participants are classified into two types: classes and patterns. Classes are individual object constructs that appear in any traditional class diagram. Patterns encapsulate a collection of classes and the associations that stem from these classes.
 - Classes.
 - Knowledge. Encapsulates the shared behavior and properties between a set of different types of knowledge.

- Patterns.
 - AnyDomain. Represents the container of a set of distinct and interrelated subjects that have a significant role in the understanding and abstraction of a particular domain.
 - AnySubject. Represents a container of a set of interrelated facts, events, experience, and situations that have a significant value in the overall perception of an area of interest.
 - AnyMechanism. Encapsulates the canonical (shared) behavior and properties that occur in different types of mechanisms, regardless of their context of applicability.
 - AnySkill. Represent the different types of abilities that any type of actor can offer according to a specific subject of expertise.
 - AnyActor. Represents the set of users or stakeholders, with certain skills, that acquire or understand a particular knowledge.

- CRC Cards. Figure 40 illustrates the CRC cards of the Knowledge Stable Analysis Pattern.

Knowledge (Knowledge)		
Responsibility	Collaborators	
Represents any type of knowledge that your system requires or collects.	Clients	Server
	AnyDomain AnyActor	acquire(), convey(), peek(), assess(), referTo()

AnyDomain (Domain)		
Responsibility	Collaborators	
Serve as the bridge between knowledge understanding and acquisition and a particular subject.	Clients	Server
	Knowledge AnySubject	define(), explore(), localize(), scale(), constrain(), disclose()

AnySubject (Subject)		
Responsibility	Collaborators	
Captures the different subjects of interest during the knowledge acquisition activity.	Clients	Server
	AnySubject AnyDomain AnyMechanism	express(), capture(), select(), assign(), stratify(), findSubject()

AnyMechanism (Mechanism)		
Responsibility	Collaborators	
Represents the different types of mechanism that will be used during knowledge acquisition.	Clients	Server
	AnySubject AnySkill	bind(), invokes(), cancel(), adjust(), report()

AnySkill (Skill)		
Responsibility	Collaborators	
Represent the different skills that an actor can have.	Clients	Server
	AnyActor AnyMechanism	obtain(), perform(), indicate(), denoteHost()

AnyActor (Actor)		
Responsibility	Collaborators	
Represents the different types of actors that will gain knowledge.	Clients	Server
	AnySkill Knowledge	act(), demonstrate(), select(), understand(), gain()

Figure 40. The CRC Cards of the Knowledge Stable Analysis Pattern

- Behavioral Model. Figure 41 illustrates the behavioral model of this pattern.

Figure 41. The Behavioral Model of the Knowledge Stable Analysis Pattern

- Consequences.
 - In distributed systems, for example, where the necessity for broadcasting and collecting relevant information over different media (e.g., physical networks, wireless networks) is important. The Knowledge Stable Analysis Pattern has the ability to adapt to this environment by just extending its boundary and adding the AnyMedia pattern to its core. This pattern was not included in the original pattern due to certain factors: knowledge exists before any

media, which means that one can have knowledge without utilizing any media; knowledge has the option to be stored or documented through any media (e.g., local inventory management systems).

- In e-commerce systems using agent technology, for example, where agents have to log the customers purchasing behavior in order to understand and gain knowledge about these customers, knowledge has the option to adapt to this new environment by simply adding the AnyLog pattern to its core. This pattern has a centralized means for recording events or accessing the IOs related to the source of the collected information. This pattern was not included in the Knowledge Stable Analysis Pattern for two reasons: instead of AnyLog, one may have decided to use a different facility to record these customers' events (via IOs); knowledge is not bound to specific means to record customers' events.
- Applicability with Illustrated Examples. The following two scenarios provide two possible usage scenarios of this pattern. For the sake of simplicity, the complete pattern's model was not included.
 - Scenario 1. Autonomic Computing Context. Briefly, autonomic computing relies on the idea of the creation of self-governing systems that can adapt and manage themselves in accordance to stakeholders' interest (e.g., automatic system installation). Self-governing systems will use gained knowledge acquired from detecting and analyzing log files, and reading system configuration

to detect and solve local problems related to bugs or logical errors. Using the gained knowledge about these possible failures, the self-governing system will look in a specific repository for the right patches (if available) and fix these problems. Figure 42 illustrates this example.

Figure 42. The Autonomic Computing Context Scenario

- Scenario 2. A Customer Relationship Management (CRM) System. In B2B, C2B, or any other systems within the e-commerce's realm, the acquisition of customers' knowledge, especially the understanding of these customers, their needs, aims, and wants, is their cornerstone. Therefore, one will need a system that facilitates the proper understanding of this customers' knowledge and allow businesses to align their processes, products, and services to build good customer relationships and increase the businesses' benefits. Figure 43 illustrates this scenario.

API (e.g., WeakHashMap). These classes will streamline the way the garbage collector will get rid of the object's memory overhead in a pattern's application.

- Don't implement the Serializable interface in EBTs for the following reasons: there is only one instance per EBT created – they are extendable singletons; Serializable interface allows the implementation of classes without the necessity of calling the class's constructors. Try to use your own custom serializer.
- BOs can implement the Serializable interface since they don't need to be unique instances in the system.

AnyContext Stable Design Pattern Detailed Template

- Name. AnyContext Stable Design Pattern.
- Known as (Optional). N/A
- Context. Context is a term that is widely used in different fields, such as computer science, linguistics, negotiations, etc. Regardless of the area of application, a context main goal is to enclose any type of relevant information that can characterize any type of entity, such as person, system, class, etc. This characterization is based on certain and varied conditions or circumstances. For instance, a canonical example of context is found in programming languages, such as the methods' scopes represented by curly brackets (e.g., methodName() {...}). Everything

within these curly brackets defines the method's context. In other words, the curly brackets encapsulate information that belongs only to the method itself. Another instance of contexts is found on clearance systems. In clearance systems a person is assigned a code that allows him/her to access certain information. The access scope of this code will represent the context of the person.

- Problem. The realization of context as a reusable asset for different solutions is a complex, most likely vague, activity that can lead to misinterpretations, constant debates, and recurrent cost/effort/time. Context is usually addressed in different ways that share the minimal, or nothing, number of characteristics and behavior per implementation. The reason for this is clearly a poor and incomplete analysis of what the structure of a context is. So, how can one capture and share an enduring structure and behavior of a context that could span several applications?
- Challenges and Constraints.
 - Challenges.
 - To provide a generic definition of context based on business and software perspectives as a whole unit.
 - To provide the means for a proper encapsulation of any entity's relevant information.
 - To cope with different types of conditions or circumstances that may qualify an entity in a systematic manner.

- To allow a context to include the definition of any type of actor, rather than relying of specific actors' roles.
- To decouple the definition of any type of entity from its actual implementation.
- To come up with the right abstractions that identify what a context is, how it is structured, and how it will behave.
- Constraints.
 - AnyActor must understand what the conditions or circumstances that qualify the entity of interest are before determining this entity's context of application.
 - AnyContext's definition must wrap its rationale (Encapsulation) within its structure, so that it could take advantage of rationale' functionality without exposing its rationale to the outside world.
 - The conditions that may qualify an entity must serve as significant inputs in the encapsulation activity.
 - One or many contexts may be determined by one actor.
 - A context can hold at least one condition that drives the encapsulation of information.
 - The conditions that are aggregated in a context are established by one or more actors.
 - One or many conditions qualify one entity of interest.
 - Entities can be composed by one or many entities.

- Solution. The solution is divided into pattern's structure and participants.
 - Structure. Figure 44 illustrates the structure of this pattern.

Figure 44. The Structure of the AnyContext Stable Design Pattern

- Participants. Participants are classified into classes and patterns.
 - Classes are individual object constructs. Patterns encapsulate a collection of classes and their associations with each other.
 - Classes.
 - AnyContext. Represents the enclosed scope of relevant information of any type of entity.
 - Patterns.
 - AnyActor. Represents the one that determines a context based on certain conditions.
 - AnyEntity. Identify the entity to be qualified and represented within context.

- AnyCondition. Represents the varied events, situations, or conditions qualifying an entity.
- Encapsulation. Represents the localization of any relevant information of any type entity.
- CRC Cards. Figure 45 illustrates all the CRC cards of this pattern.

AnyContext (Context)		
Responsibility	Collaborators	
Represents any type of context	Clients	Server
	AnyActor AnyCondition AnyEntity Encapsulation	determine(), characterize(), encapsulate(), enclose(), disclose(), associate()

AnyEntity (Entity)		
Responsibility	Collaborators	
Represents the different types of entities that can be enclosed in a context.	Clients	Server
	AnyCondition AnyContext AnyEntity Encapsulation	compose(), separate(), portray(), behave(), referTo()

AnyActor (Actor)		
Responsibility	Collaborators	
Represents the distinct types of actors interacting in a context.	Clients	Server
	AnyCondition AnyContext	acknowledge(), acts(), access(), provides()

AnyCondition (Condition)		
Responsibility	Collaborators	
Represents the distinct types of conditions qualifying any type of entity.	Clients	Server
	AnyEntity AnyContext AnyActor	establish(), qualify(), apply(), evidence(), isValid()

Encapsulation (Encapsulation)		
Responsibility	Collaborators	
Localize the relevant information about any type of entity.	Clients	Server
	AnyEntity AnyContext	localize(), publish(), constrain(), protect()

Figure 45. The CRC Cards of the AnyContext Stable Design Pattern

- Behavioral Model. Figure 46 shows the pattern's behavioral model.

Figure 46. The Behavioral Model of the AnyContext Stable Design Pattern

- Consequences.
 - Since contexts are ruled by different set of conditions, determined by different sorts of actors and entities, and deployed over different environments, it will be required to use a different set of new patterns to turn this pattern more application specific. Examples of these patterns are AnyLog pattern, in the case, for example, one need to record the activities of an actor. Another example is found in areas such as data mining, ontological engineering, and component testing, etc. In this case, it will be required to integrate to the current solution the AnyMedia pattern, etc.
- Applicability with Illustrated Examples. The following two scenarios provide two possible uses of this pattern. For the sake of simplicity, the complete pattern's model was not provided.
 - Scenario 1. Context Sensitivity Customization in Component-Based Systems (CBS). The idea is to create a system that will assure a proper context sensitivity customization of component-based applications. This customization will be realized by the systematic propagation of calling and deploying contexts, along with the provided contextual properties, within the component-based system. Figure 47 illustrates this scenario.

Figure 47. The Context Sensitivity Customization in CBS Scenario

○ Scenario 2. Context-based Storage System in Mobile

Environments. The proposed systems will facilitate the access and management to stored information from users across mobile environments based the users' involved situation or state. For example, the system will facilitate relevant information to the user according to the users' rights or permissions with respect to the accessed information, the environments from where the user is accessing the information, the available bandwidth, etc. Figure 48 illustrates this scenario.

Figure 48. The Context-based Storage System Scenario

- Tips and Heuristics.
 - Design Heuristics.
 - EBTs can be aggregated into BO, but not the other way around. This will prevent creating strong dependencies between EBTs and BOs.
 - Coding Heuristics.
 - To determine performance of one's methods calls, a programmer can implement a set of micro-benchmarks that will allow him/her to determine performance changes of her/his class's methods over time (after attaching new IOs).

APPENDIX B. Source Code

Frameworks' Main Interfaces or Types

The following snippets provide the implementation of the main types of the Knowledge Maps Framework. The additional types, such as Analysis and Design will be provided as well. Regarding the Hooking Facility and the rest of classes, the source code will be provided in a separate media. Snippet 10 shows the type definition of the EBT type. This type definition consists of 5 interfaces.

```
package net.hsanchez.km.core;
public interface EBT {
    /**
     * @return String
     */
    String introduces ();
    /**
     * Enlist a BO into the EBT's registry
     * @param member
     */
    void enlist ( BO member );
    /**
     * Releases a BO from the EBT's registry
     * @param member
     *      BO to be enlisted
     * @return
     *      <code>True</code> it was released
     *      <code>False</code> it was not released
     */
    boolean releases ( BO member );
    /**
     * Divides an EBT into a set of subgoals
     * @param subgoal
     *      subgoal to be added
     */
    void divides ( EBT subgoal );
    /**
     * ignores a subgoal from the EBT's subgoal registry
     * @param subgoal
     *      Subgoal to be added
     * @return
     *      <code>True</code> it was ignored
     *      <code>False</code> it was not ignored
     */
    boolean ignores ( EBT subgoal );
}
```

Snippet 11. EBT Type

Snippet 11 shows the type definition of the BO type. This type definition consists of 10 interfaces.

```
package net.hsanchez.km.core;
public interface BO {
    /**
     * @return
     */
    String introduces ();
    /**
     *
     * @param goal
     */
    void collaborates ( EBT goal );
    /**
     * @param collaborator
     */
    void connects ( BO collaborator );
    /**
     * @param collaborator
     * @return
     */
    boolean disconnects ( BO collaborator );
    /**
     * @param extension
     * @return
     */
    BO loads ( String extension );
    /**
     * @param extension
     */
    void disables ( String extension );
    /**
     *
     * @param extension
     */
    void attaches ( ExtensionPoint extension );
    /**
     * @param extension
     * @return
     */
    boolean removes ( ExtensionPoint extension );
    /**
     *
     * @param industry
     * @return
     */
    IO uses ( IO industry );
    /**
     * @return
     */
    boolean isPersistent (
    );
}
```

Snippet 12. BO Type

Snippet 13 shows the type definition of the ExtensionPoint type. This type definition consists of 13 interfaces.

```
package net.hsanchez.km.core;
public interface ExtensionPoint {
    /**
     * Returns the name of the ExtensionPoint
     * @return String
     */
    String describes();
    /**
     * Add a BO to the ExtensionPoint
     * @param member
     *         BO to be added
     */
    void binds(BO member);
    /**
     * Returns the Type of ExtensionPoint
     * @return String
     */
    String getType();
    /**
     * Add an IO to the ExtensionPoint's IOs registry
     * @param io
     *         IO to be added
     */
    void add(IO io);
    /**
     * Remove the IO from the ExtensionPoint's Ios registry
     * @param io
     *         IO to be removed
     * @return
     *         <code>True</code> It was removed
     *         <code>False</code> It was not removed
     */
    boolean remove(IO io);
    /**
     * Set the Parent ExtensionPoint
     * @param extension
     *         Aggregated ExtensionPoint
     */
    void setExtensionPoint(ExtensionPoint extension);
    /**
     * Returns ExtensionPoint type
     * @return ExtensionPoint
     */
    ExtensionPoint getExtensionPoint();
    /**
     * Set the AbstractExtension version to the ExtensionPoint concept
     * @param aep
     */
    void setAbstractExtension(AbstractExtensionPoint aep);
    /**
     * Get the AbstractExtension version of the ExtPpoint Type
     * @return AbstractExtension
     */
}
```

```

    */
    AbstractExtensionPoint getAbstractExtensionPoint();
    /**
     * Enables the ExtensionPoint
     */
    void enables();
    /**
     * Disables the ExtensionPoint
     */
    void disables();
    /**
     * Is a valid ExtensionPoint?
     * @return
     *     <code>True</code> ExtensionPoint generation has resulted no valid
     *     <code>False</code> ExtensionPoint generation has resulted invalid
     */
    boolean isValid();
    /**
     * Is the ExtensionPoint disabled?
     * @return
     *     <code>True</code> The extPoint is disabled
     *     <code>False</code> The extPoint is not disabled
     */
    boolean isDisabled();
}

```

Snippet 13. ExtensionPoint Type

Snippet 14 shows the type definition of the IO type. This type definition consists of zero interfaces. It will be simply used as an indicator that a class is an IO.

```

package net.hsanchez.km.core;
public interface IO {
}

```

Snippet 14. The IO Type

Snippet 15 shows the type definition of the Analysis type. This type definition consists of 7 interfaces.

```

package net.hsanchez.km.artifacts;

import net.hsanchez.km.core.EBT;
import java.util.Date;

public interface Analysis extends EBT {
    /**
     * Obtain the Identifier of the Analysis Concern
     * @return long
     */
    long getID        ();
    /**
     * Determine if the Analysis concern is active
     * @return
     */
}

```

```

*      <code>True</code>
*      <code>False</code>
*/
boolean      isActive      ();
/**
 * Returns the date when this concern was created/instantiated
 * @return java.lang.Date
 */
Date  getCreatedDate ();
/**
 * Returns the type of analysis concerns we are dealing with
 * @return java.lang.String
 */
String getType      ();
/**
 * Returns the current domain where this concern is being used
 * @return java.lang.String
 */
String getDomain      ();
/**
 * Satisfy the Analysis Concern
 */
void  satisfy      ();
}

```

Snippet 15. The Analysis Type

Snippet 16 shows the type definition of the Design type. This type definition consists of 4 interfaces.

```

package net.hsanchez.km.artifacts;

import net.hsanchez.km.core.BO;

public interface Design extends BO {
    /**
     * Returns the Design concern's identifier
     * @return int
     */
    int  getID      (
    );
    /**
     * Returns the current Design concern's responsibility
     * @return java.lang.String
     */
    String getResponsibility (
    );
    /**
     * Determines if the design concern is enabled
     * @return
     *      <code>True</code> The design concern is enabled
     *      <code>False</code> The design concern is not enabled
     */
    boolean  isEnabled      (
    );
    /**
     * Clone the Design concern for any unknown reason
     * @param capabilityID
     *      The identifier of this design concern
     */
}

```

```
* @return java.lang.Object
*         The design concern instance
* @throws CloneNotSupportedException
*         if there is a unknown error, fire this exception
*/
Object clone (int capabilityID) throws CloneNotSupportedException;
}
```

Snippet 16. The Design Type