UNIFIED SOFTWARE ENGINEERING REUSE: A METHODOLOGY FOR EFFECTIVE SOFTWARE REUSE

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Charles Flood

May 2017

The Designated Thesis Committee Approves the Thesis Titled

UNIFIED SOFTWARE ENGINEERING REUSE:
A MEANS OF MAXIMIZING SOFTWARE REUSABILITY

by

Charles Flood

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

May 2017

Mohamed Fayad, Ph.D.	Department of Computer Engineering
Dan Harkey, MS	Department of Computer Engineering
Xiao Su, Ph.D.	Department of Computer Engineering

ABSTRACT

UNIFIED SOFTWARE ENGINEERING REUSE:
A METHODOLOGY FOR EFFECTIVE SOFTWARE REUSE

by Charles Flood

Software is a necessity in the modern world, and that need is continuously growing. As expensive as the creation of all this new software is, the maintenance costs are even greater. One solution to this problem is software reuse, whereby already written software can be applied to new problems after some modification, thus reducing the overall input of new code. The goal in traditional software reuse is to produce a piece of software with enough flexibility to be used at least twice. Unfortunately, there are many difficulties in achieving software reuse using modern programming techniques. Even software built specifically for reuse is severely restricted in its utility for new applications. It is easy for new programs to require entirely new logic or new objects. Because of this, they become quickly outdated, and any labor spent creating reusable software is nullified. The solution is a method to vastly increase the reusability of software by concentrating on the base knowledge and overall goals of software rather than the details on a case-by-case basis. Finding patterns in the problem and solution spaces allows unification into a smaller solution set. Instead of each problem receiving its own solution from marginally reusable components, multiple problems are resolved with the same architecture and object set. As an added benefit, this solution will not only vastly improve software reuse, but it will make feasible systems that can construct software architecture on demand and provide the first steps to fully automated software development.

ACKNOWLEDGEMENTS

For his substantial guidance and sacrificed time in the production of this thesis and all the associated processes, Dr. Fayad has secured my enduring gratitude.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API – application program interface
BO – business object
CBSD – context-based software development
EBT – enduring business theme
IO – industrial object
SSM – stability software modelling
USER – unified software engineering reuse

Chapter 1: Stable Software Reuse Overview

Twenty-five years ago, the notion was you could create a general problem-solver software that could solve problems in many different domains. That just turned out to be totally wrong.
— Howard Gardner (Koch, 1996, p. 57)

Introduction

Effective software reuse has long been one of the goals of computer specialists and businesspeople alike. Even now, this goal remains elusive as current reuse schemes by their nature tend to have exceedingly narrow applications. Furthermore, correctly implementing those techniques requires a skill and patience that are all too often lacking.

Alternatively, through unification of several techniques, practically unlimited reuse can be achieved, yielding assets that will be externally adaptable to virtually any scenario. This is done through the analysis of generalized scenarios to extract and model the core knowledge common to that aspect of reality. Employing this and other techniques allows for the creation of near-infinitely reusable assets.

When one discusses software reuse in a college classroom or a business meeting, the conversation is inevitably about traditional software reuse (Amar & Coffey, 2005; Capiluppi, Stol, & Boldyreff, 2013; Constantinou, Ampatzoglou, & Stamelos, 2014; Coulange, 2012; Ezran, Morisio, & Tully, 2002; Mili, Mili, Yacoub, & Addy 2001; Mojica et al., 2014). Traditional reuse places a strong dependence on object orientation, software patterns, and the skill of the developer to ensure that any software developed will be as flexible and long lasting as possible. This is done in the hopes that future maintenance of the software will be less costly if it is designed and built to handle potential changes. Unfortunately, software maintenance still accounts for a staggering percentage of software costs in the business world (Galorath, 2008).

Traditionally, each piece of software is a solution to a specific problem. Creating a solution for every problem is a simple and straightforward way to ensure the developer solved the problem. This results in equally infinite problem and solution spaces. Such one-to-one mapping is not feasible for such a large domain. The traditional way to handle this situation is to pare down the domain by only creating software for a few chosen scenarios. The alternative, thus far mostly ignored, is to constrict the range. By mapping multiple problems on the same solution and thereby achieving software reuse, one can vastly increase the number of successfully solved problems.

Unified software engineering reuse (USER) is built on this concept of unifying multiple problems into a single overarching problem with an equally singular solution. This can be accomplished using stable software modeling to create infinitely reusable software assets that can be combined into stable software patterns. Each pattern is a solution to a set of problems as opposed to a solution to a single problem and can be combined with other patterns to solve ever more complex problems. As long as the root problem is clearly defined and the assets exist, USER will offer an effective solution, and software production costs, both temporal and monetary, will be significantly reduced.

Potential Problems and Pitfalls of Existing Software Reuse

It should first be noted that software reuse already exists in a limited form, but like any technology, investments of time and energy are required for it to mature. The current forms of software reuse have a number of obstacles for designers to overcome (Nurolahzade, Walker, & Maurer, 2013; Kulkarni & Varma, 2016; Schmidt, 1999).

Mentioned here are just a few of the potential pitfalls for software engineers attempting to build reusable software.

Experience required. One of the greatest issues with the current means of existing software reuse is that it requires substantial experience to implement adequately. (Morisio, Ezran & Tully, 2002) Careful study of specific techniques is required, and many mistakes are made while the programmer slowly gains experience. Although there is no substitute for hard work in any endeavor, a decade's experience seems an excessive price for reuse.

Finding the right tools. Software reuse presents a challenge even for experienced software developers. Locating the right components to reuse and properly adapting them to the software remain difficult especially in more abstract contexts. Software libraries with well documented application program interfaces (APIs) are often simple to integrate into a piece of code but grafting entire pieces of software into an enterprise level system is another matter entirely. All too often, the software performs inadequately or fails to yield the proper outcomes because it was not originally designed with the new context in mind. Overcoming these difficulties is only the first step because once the software is complete, the system will require ongoing maintenance. The software components require updates that may cause them to deviate from expected functionality in the system.

Scheduling. Developing good software is not particularly difficult or stressful to software engineers as long as an infinite amount of time is available. However, time constraints are a very real part of every major software-related undertaking. This adds pressure on the developers to design and build quickly, which could easily lead to less flexible or poorer quality software.

Complexity. There is an acceptable level of complexity in any given piece of software often directly corresponding to the complexity of the problem being solved. It is accepted that reusable software is necessarily more complex to accommodate the extra flexibility. This runs counter to a very real need to develop simple code that is easily maintained or altered. In this sense, making software more reusable could make it less so, especially when poorly done.

Probable Solutions

When incorporating USER, the pitfalls of traditional software engineering mentioned above, and many others, are either fully or partially solved. While developing a stable software pattern does require some experience, using a pattern to build software does not. No external tools are needed because USER includes the tools. As long as the required patterns exist, software can be created on demand, allowing the developer to easily produce the software before any deadlines. Finally, although USER-based software contains more classes and code than traditional reusable software, it is less complex because the models make reading and understanding each program exceedingly simple.

Overview of Stability Software Models

Stability software modeling (SSM) is a major component in USER. It is based on the idea that every program has an ultimate objective that transcends space, time, and the nature of software itself. These enduring business themes (EBTs), such as friendship, greed, ownership, and order, are the ultimate goals in a number of real-life and software scenarios. For example, the whole point of an asset is ownership. If a piece of software needed an asset object, all of the objects associated with an asset, as well as its EBT (ownership), would be included in the architecture of the program.

After EBTs, the next layer is business objects (BOs). Each BO is an abstraction of a physical object or an idea. That does not mean that BOs are sometimes tangible objects; in fact, none are. An asset, for example, can be any of a number of physical objects, but it is not implicitly tangible. Likewise, a schedule is an abstraction of an idea. Unlike EBTs, BOs do not transcend space or time. A schedule has a start and an end. It can have multiple instances; while love, joy, and peace exist without end or instantiation.

The final layer in SSM is the industrial objects (IOs). These are the actual objects that most programmers would identify in the program. For example, air traffic control software without SSM would probably include objects like planes, runways, a tower, and a controller. With SSM, each of these would be IOs. It is important to note the temporary nature of IOs, both in the sense that individual objects may cease to be and that the object class itself may one day be obsolete.

With just the EBTs and BOs it is possible to create the core functionality of any scenario and to then apply it to other similar scenarios. This allows the developer to hook on any IOs that are required for a given scenario, providing needed flexibility. In the air traffic control software mentioned above, the same software implemented in SSM would be equally capable of handling ships in a harbor, or intergalactic spacecraft at a space station. The core software would remain as new IOs are hooked into place to make a new scenario functional.

Overview of USER

USER optimizes the reusability of software by unifying a large set of problems into a single problem through abstraction. For example, influence can be found in many places, such as in online or print advertising and in big data analytics for business and politics.

The potential scenarios in which a model for influence would be useful are many and varied, but since they all share common ground by using influence, they can also share a common software solution. The developer of such a solution creates many solutions through software reuse.

Contributions

More is presented in this thesis than just the concept of unlimited reuse through unification of problem sets. It also includes a starting set of patterns and knowledge maps to begin implementation once analysis is complete. The knowledge maps for software reuse and context will aid in an overall understanding of the concepts, what their objectives and requirements are, and how to best employ them.

A number of stable design patterns will be presented in this thesis as well.

- Any schedule stable design pattern
- Any influence stable design pattern
- Any stress stable design pattern
- Reputation stable analysis pattern
- Conflict resolution stable architecture pattern

While some of these patterns will merely be used as examples to compare USER with traditional software modeling, others will be examined in detail as a means of analyzing the specific problems that they solve and analyzing the nature of USER and the way it solves problems in a general sense.

Significance

The significance of USER is its potential to revolutionize the entire field of software. With limitless reuse, software can be written in far less time and for a fraction of the cost and still be equally or more effective than software written using modern techniques. As a result, the need for programmers would diminish as it is replaced by the need for more information engineers.

Summary

In this thesis, the premise that a new method for producing reusable software far superior to current techniques is presented. Through unifying similar problems into one and applying knowledge to the resulting problem, software may achieve a level of reusability heretofore unimagined.

Chapter 2: A Comparative Study between Existing Software Reuse and USER

"Let us search and try our ways…" — Lamentations 3:40 (KJV)

Introduction

In software reuse, the traditional methods lead to convoluted models and code when they are implemented, ultimately yielding artifacts that are seldom reused as desired. A new, more effective solution to the problems posed by traditional software reuse should then be supported by a superior model. In this chapter, a comparative study of the two software reuse techniques will be used to demonstrate which model is the more efficient, while also being easily implemented in future work.

Abstractions and Levels of Abstraction.

A major part of software reuse is abstraction. This is what allows us to adequately model anything in software. For example, when creating software for a self-driving car, it is useful to have software classes and objects such as car, wheel, engine, and brake. Since these are physical things to be represented in software, this is the first layer of abstraction.

With USER, each object is a part of a larger whole. If an architecture has 20 objects, each of those objects has its own subpattern that defines its behavior. This is the second layer of abstraction added to the software (Hamza & Fayad, 2002).

Defining Traditional Software Reuse

Systematic software reuse may be defined as developing software from a collection of building blocks that leverage similarities in requirements, architecture, or design (Ezran, Morisio, & Tully, 2002). Conceptually, it is easy to understand that if the same problem emerges multiple times in similar but different problems, a finite unit of software can be used in each of the several places it is needed. This is the primary essence of conventional reuse, which tends to have a number of facets or requirements, such as:

- Understanding how reuse contributes to business goals,
- Defining technical and managerial strategies to achieve maximum value,
- Integrating reuse into the software development processes,
- Ensuring staff have the necessary competence and motivation,
- Establishing appropriate organizational, technical, and financial support,
- Using appropriate measurements to control reuse performance.

Admittedly, meeting these objectives requires dedication on the part of software engineers, nor is it necessarily advisable for all engineers in an organization to pursue these goal simultaneously. Some of these goals would be better achieved by a project manager. In addition, there is a distinction between reusing and supporting the reuse. Integration is the most difficult for a manager to ensure as well as the most necessary. Therefore, the manager must create a system in which reuse is already a built-in and foregone conclusion.

Existing Software Reuse Types (Building Blocks)

Conventional reuse is predicated on the existence of building blocks from which any application is built. Given the ubiquity of object oriented everything in software engineering, the concept of building blocks is equally ubiquitous and unambiguous. Proper building blocks tend to have several properties of their own.

- They can be used and combined to create new building blocks;

- They may or may not be designed to function as building blocks;
- They may or may not be designed to fit in a certain way;
- A larger pool of building blocks yields more diverse construction options;
- Special building blocks are less likely to be used;
- Large building blocks may be designed to fit smaller ones inside (Ezran, Moriso, & Tully, 2002).

Conceptually, these are the equivalent of any modular toy construction system, such as Lego, K'nex, and Erector.

Stable Software Reuse

Stable software reuse takes the concept of systematic software reuse and applies it more holistically to the realm of software. It no longer considers the code base alone but also the models, contexts, documentation, tests, and every other element of the software design process. The objective is no longer to tailor objects to a specific application but to make a universal set of software artifacts that can be used in an infinite number of settings. With this change in scope comes a change in definition and attributes.

The first change concerns the strategies of creating a stable artifact or developing a system based on stable artifacts. The initial properties given above no longer apply under this new definition. Stable software reuse will always provide the same advantages, including improved productivity and maintainability, so that an understanding of the relationship between the instance of reuse and its goals are no longer necessary.

The specific strategies involved are a part of the system for stable software reuse and no longer need to be individually developed on a per project basis:

- Integrating reuse into the software development processes
- Ensuring staff have necessary competence and motivation
- Establishing appropriate organizational, technical, and budgetary support
- Using appropriate measurements to control reuse performance

Some of these strategies, such as staff competence and motivation, are still essential to business in general, but these aspects are not necessary exclusively for reuse. Likewise, appropriate support must be considered as it relates to the entirety of the business. However, the needed support for stable reuse is trivial compared to the organization as a whole.

Stable Software Reuse Types

There are a number of characteristics for stable software reuse. However, some of these are important for any project.

Artifacts. Artifacts are the end results of stable software development prior to using the software in the field. These could be models, patterns, contexts, or any number of things that are generated as the primary product of the software stability process. The defining feature of a stable software artifact is its ability to be reused in any number of circumstances. These artifacts are analogous to the building blocks of traditional reuse, but they can encompass such reusable things as documentation, requirements, use cases, or test cases.

Patterns. In SSM, patterns can be divided into three types, depending on the nature of the central element of the pattern. The two simpler patterns, Stable Design Patterns and Stable Analysis Patterns, focus on single concepts, while Stable Architecture Patterns are more complicated.

Stable design patterns. Stable design patterns revolve entirely around abstracted objects, but they are never representations of physical objects. For example, a schedule design pattern focuses on a generic schedule and not a specific employee schedule. The design pattern keeps this object at its core while seeking out the ultimate purpose for the existence of that object (for a schedule, it is coordination). Once the purpose of the object is known, the pattern can be completed with other generic objects that help the object support that purpose. From there, the pattern can be integrated with others, or used on its own as a design for a software system.

Stable analysis patterns. Stable analysis patterns always focus on a single enduring concept, such as friendship, respect, or unity. For these patterns, the objective is to analyze the concept and find the real-world components that are essential to the concept. By abstracting these real-world objects, it leaves a pattern that describes the concept universally. This pattern makes it easier to interpret situations in real-life and find missing factors that lead to solutions. The pattern may be used alone to create software, but it is far more likely to be combined with other patterns first.

Stable architecture patterns. By combining multiple patterns, whether design or analysis, it is possible to create stable architecture patterns. For example, the pattern for conflict resolution presented in a later chapter is an architecture pattern based on the conflict design pattern and the resolution design pattern. By combining the concepts and elements of multiple patterns, one can gain a broader range of possible software solutions that perform with multiple requirements.

Contexts. A single, conventional program will need a plethora of use cases to accomplish its requirements, development, and associated documentation. While one may acknowledge the value in clarifying the expectations of software, these use cases are seldom reusable for similar applications. However, one can abstract the software one level up and make the conventional program or make the scenario for which it was designed a context. The stable software is constructed with a finite number of contexts or scenarios in mind, but the number of potential contexts that can arise is practically infinite. This would be a major concern were it not for the fact that stable software by design utilizes the core knowledge and concepts of the system it is built to describe. Accordingly, unless there is a change that unseats the entirety of a system of concepts, stable software will be able to handle any new contexts as they arise.

Documentation. While not commonly considered when discussing reuse of software, it is only reasonable that the documentation of the software itself also be reused. This frees the software engineer from the requirements of delicately crafting easily understood, yet concise and technical documentation. In current software engineering, self-documenting software is available, such as with Javadoc, but only if the programmer takes the time to add specifically pre-formatted comments to the code. Even then, these systems are optimized primarily for the creation of class APIs. With USER, artifacts, components, patterns, and their documentation can be reused in equal measure.

Comparative Study Weighted Criteria

To adequately compare individual instances of current software against USER designed systems, criteria must be defined for that judgment. Since all commercial software must meet certain quality standards, those measured standards should make an adequate list from which to begin a comparative study (Kan, 2002). These are given in Table 1.

Table 1

Software analysis criteria for comparing models.

Criterion	Weight	Reason
Reusability	.20	Poor reusability puts greater strain on support and requires better documentation to compensate. Ensuring good reusability reduces headaches and software production costs.
Simplicity	.20	Since software cost comes primarily from maintenance, software should remain as internally simple as possible to help reduce costs.
Completeness	.10	A program that does not handle all use cases can prove frustrating to clients and end users. Since this factor often only impacts the subset of users, less significance is granted to it than for usability.
Testability	.10	This factor represents the ease with which developers and testers can create tests for the software. This usually correlates to the simplicity factor.
Extensibility	.10	Extensibility reflects the ease with which new functionality can be added to the software.
Stability	.10	Stability refers to the ability of the software to adapt to changes in business seamlessly. The more stable the software is, the less it will need to be replaced and the easier and cheaper it will be to maintain.
Portability	.05	Portability refers to a software's ability to operate across a wide array of platforms. This has become even more difficult in recent times as mobile devices differ greatly from standard PCs. Not all applications need to be ported to other platforms, but the added flexibility is worth pursuing, if practical.
Scalability	.15	Most commercial applications need to store and process large datasets across multiple machines.

Comparative Study

An illustration is the easiest way to show the comparison between traditional software reuse and USER. Figure 1 shows a UML class diagram for a generic schedule based on USER principles.

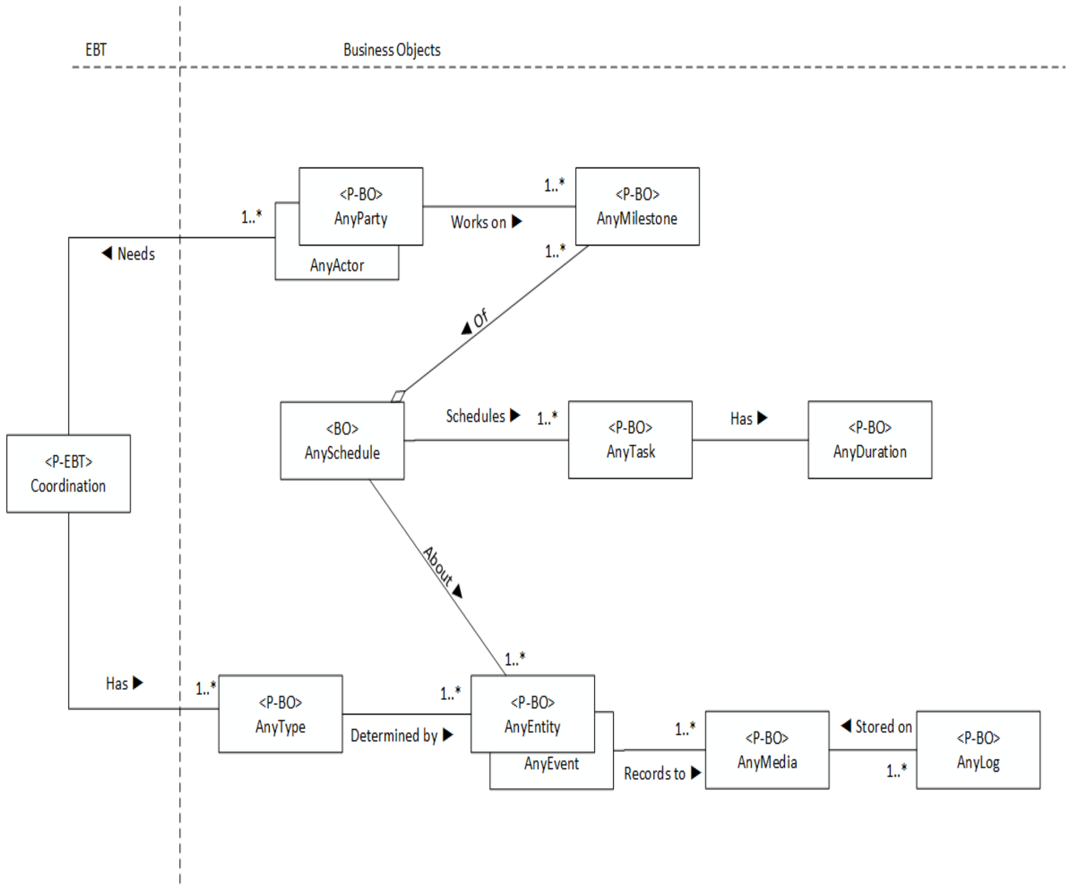

Figure 1. SSM for any schedule based on USER principals.

This diagram can be compared to a simplified design for an application created to track employee schedules in a business as in Figure 2. In this more conventional model, classes are less abstracted, which makes them easier to initially conceptualize, but less flexible.

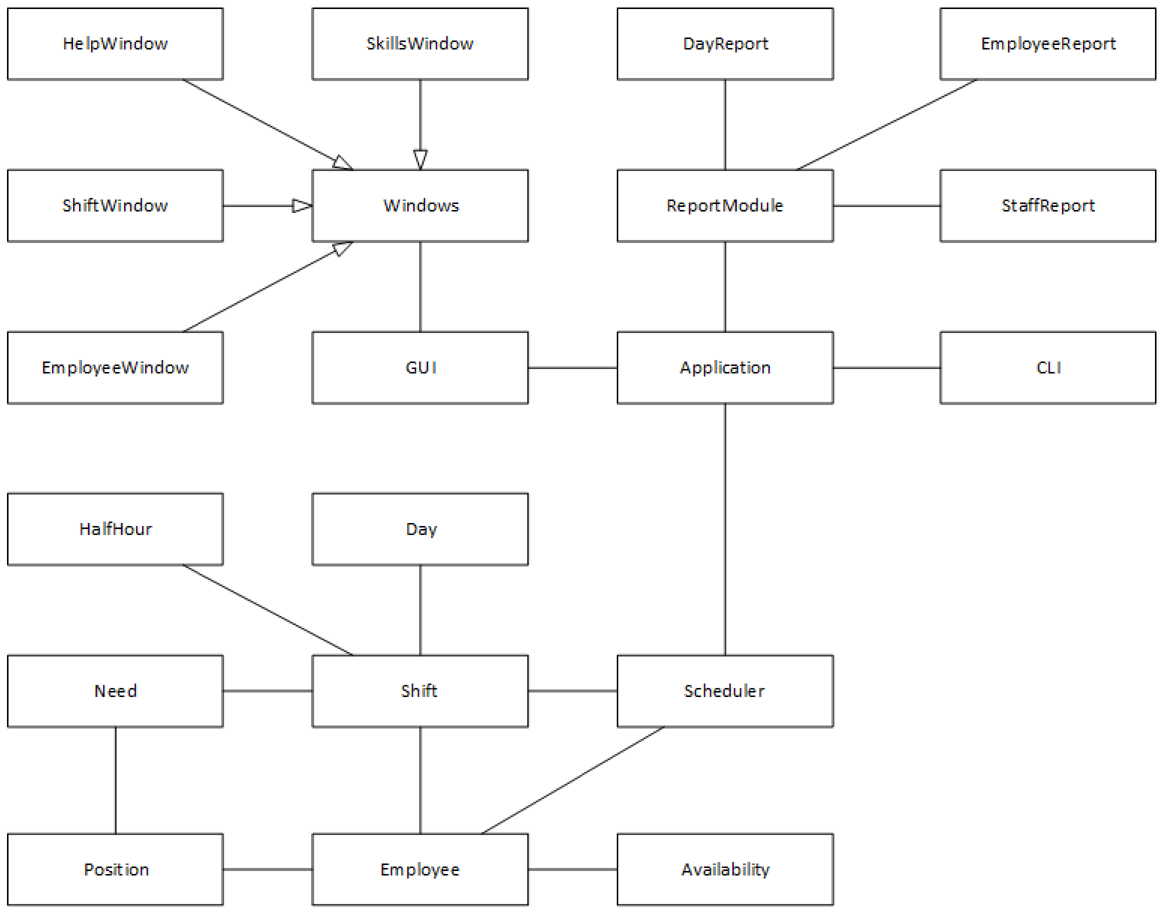

Figure 2. A traditional model for a specific work schedule.

Analysis and Discussion

Based on the weighted criteria from Table 1, which is the better application? The USER pattern requires fewer transition points making it simpler than the traditional model. While it seems the traditional model is more complete, the USER pattern either directly contains all the elements or those present can be easily extended. This brings forward the fact that USER is the natural winner in extensibility, as it is capable of handling more than just employee schedules. This flexibility also gives USER the edge in stability. Scalability is the last remaining factor, but both perform equally well there. A tabular format of this analysis is given in Table 2.

Table 2

Criteria of traditional vs. USER model of schedule.

Criteria	Weight	Traditional Score	USER Score	Reasoning
Reusability	.20	.02	.20	The USER model is inherently reusable, while the traditional model provides limited reuse
Simplicity	.20	.05	.20	The USER model is cleaner and easier to understand at a glance
Completeness	.10	.10	.10	Both models are complete
Testability	.10	.00	.10	The traditional model must be implemented before testing can begin, while the USER model can have tests applied now.
Extensibility	.15	.00	.15	The traditional model only works in a multi-shift business.
Stability	.10	.00	.10	Small changes to the business could necessitate a full rewrite.
Scalability	.15	.03	.15	The traditional model is only able to scale up
Total	1.00	.20	1.00	

Summary

While there are some similarities between traditional reuse and USER, the differences are far greater. Traditional software is intertwined with business, and its building blocks are exceedingly vague without offering the potential for reuse of major software assets like documentation and test cases. USER not only offers this, but it also allows for a better understanding of the concepts behind the software systems. Finally, it makes the system applicable to a broad range of scenarios with little alteration.

Chapter 3: Software Reuse Knowledge Map

"The goal of software reuse is to reduce the cost of software production by replacing creation with recycling." –Yijun Yu

Introduction

The term software reuse can be defined as the process of developing software systems from existing software instead of creating them from scratch. In most software engineering disciplines, systems are designed by integrating existing software components that have been used in other systems into a new system. Software reuse has become a topic of much interest in the software community due to its potential benefits, which include increased product quality and decreased product cost in development and maintenance. To a great extent, existing software documents (source code, design documents, etc.) are copied and adapted to fit new requirements. Classically, software engineering has been more focused on original development, but it is now recognized that to achieve better software in a time-efficient and cost-effective way, a design process that is based on systematic software reuse is essential.

This chapter aims at applying the SSM approach toward creating a model for software reuse which can be applied universally. The software stability ensures high reusability, stability and a more design-efficient, domain-independent model. The key contribution is the presentation of stable pattern analysis and the listing of the EBTs and BOs involved in the area of software reuse. Such generic models can further be applied to any possible scenario.

Among software developers, waste traditionally has been encouraged as a normal part of the one-of-a-kind system development philosophy. The acceptance of waste is upheld in the name of good software practices that put user requirements first. The software tradition is to serve the customer by custom-building from scratch each system in a way that is specifically designed to meet a set of particular customer requirements.

Demand for more complex and technically evolved software applications with greater and more efficient content has long been growing at a significant rate (Myers, 1978). Of late, the software market has witnessed diverse varieties of applications that cater to an equally diverse number of industries and businesses. However, software production methods are not evolving at a similar rate.

Software developers also feel the need for improved time-to-market rates, better quality, and enhanced productivity in their daily operations. The drive to keep the development costs down forces them to look for more innovative methods that could significantly improve the design process of software applications. Although different solutions have been proposed and followed, most follow a single solution approach that seriously hinders productivity cycles.

One of the suggested software designing methods is the software reuse method. This simple, yet powerful vision was considered as early as the 1970s, though it was a byproduct of other software design strategies (Teichroew, Hershey, & Yamamoto, 1978; Walters & McCall, 1979).

Reuse of software is based on a simple, well-known idea. When a developer builds a new firmware or software application, reusing previously developed software components will save in time and budget. The cost of developing, testing, documenting, and maintaining multiple copies of essentially similar software is lower than if the software was entirely unique. Although reuse is often regarded as applying only to system components, there are many different ways to reuse. Reuse is possible for a range of levels from simple functions to complete application systems. A traditional view of a reusable software system approach was rooted in the creation of software libraries that contained generic and reusable components, which could be combined to design new software systems. Often, searching these libraries formed the basis of traditional reusability research. In effect, reusable software research utilized existing reusable resources that were considered atomic building blocks. These were eventually indexed, organized, and combined by using well-defined rules and regulations.

The traditional method of creating a new software application followed an approach that required a considerable quantity of new code written over time. This naturally had a negative impact on code quality, as well as overall cost and time for development. The simplest method to prevent such an occurrence was to write less code to reduce the time and money required to create a new software application.

It does make sense to gather and accumulate available software components from a library and reuse them to write a new application. Developers who increased the number of newer software products by using an already existing library of code could easily improve cost, time, and quality parameters. In general, the reuse approach of creating software products was a well-devised strategy for developers and enabled them to follow the current market trends that demanded technical products with faster turnaround rates.

Abstraction plays a central role in software reuse (Krueger, 1992). Concise and expressive abstractions are essential if software artifacts are to be effectively reused. Software reuse involves reuse of existing assets in some form within the software product development process. More than just code, assets are products and byproducts of the software development life cycle and include software components, test suites, designs, and documentation. Because reuse implies the creation of a separately maintained version of the assets, it is preferred over modifying existing assets as needed to meet specific system requirements. Systematic software reuse is a promising means to reduce development cycle time and cost, improve software quality, and leverage existing effort by constructing and applying multi-use assets like architectures, patterns, components, and frameworks. There are several ways in which software reuse can be achieved, such as:

- Application System Reuse—reusing an entire application by incorporating one application inside of another or developing application families (e.g., MS Office Suite);
- Component Reuse— reusing components (e.g., subsystems, single objects) from one application in another application;
- Function Reuse—reusing software components that implement a single well-defined function.

Pitfalls of Traditional Software Reuse

There are many challenges and problems that naturally arise when using traditional techniques of software development:

- High maintenance costs—maintenance is an inevitability of traditional software development. This is accepted in the business world to the extent that there exists at least one company making the analysis of software maintenance costs a major part of its business (SEER for Software - Estimating Software Development & Maintenance Costs, Version 7.3). While there are several options for a company seeking to reduce their maintenance costs, many revolve around business techniques rather than technological progress.

- Lack of source—most proprietary software is closed source. While this is a fiscal blessing for a third party producer, any consuming company is at the mercy of that third party in ensuring that recent updates, or the lack thereof, do not cause incompatibilities with the system as it evolves.
- Not-invented-here syndrome—some software engineers prefer to re-write components themselves. These engineers believe that they can improve on the reusable component or they seek to avoid reliance on third party technology because of the legal hassle associated with it.
- Creating and maintaining a component library—populating a reusable component library and ensuring the software developers can use this library is expensive. Current techniques for classifying, cataloging, and retrieving software components are immature.
- Finding software components—software engineers must be able to locate software components that will do the tasks that they need. Often an Internet search will provide good results, but even then it is not always easy to locate a deployable piece of software especially in the case of unique environments or unmaintained software.
- Adapting reusable components—very rarely will software components do exactly what they need to off-the-shelf. Components must be well documented and adapted for each situation to which they will be applied. Adapting and

reconfiguring software components for complex systems often requires a high level of expertise to first understand and then alter the software.

Properties of Software Reuse

Software reuse, whether traditional or otherwise, has a number of properties such as simplicity and ease of use (Hamza & Fayad, 2002). However, there are a number of other properties that truly reusable software should exhibit. These are seldom found in traditional approaches to software reuse. Nevertheless, the value of the properties listed here should be self-evident.

- Unification—any non-trivial software will involve several different components and have a number of requirements, each with its own goal. Well-constructed software must combine all of these disparate components and unify them into a single unit. In larger software, the goals of the many units must be unified into a single goal which the software will meet.
- Unlimited reusability—this is the ideal for any software system. With unlimited reuse an engineer can put in effort one time and continuously reap the rewards. An asset with unlimited reusability can be used practically anywhere a reasonable application exists. Such an asset does not confine itself to a single program but is applicable in a number of programs.
- Unlimited applicability—universally applicable assets are what most people refer to when discussing software reusability. Assets with unlimited applicability are those that are useful in widely differing circumstances and contexts with a minimum of change (preferably none). For example, the schema for a user profile on a social media site could be just as effectively used in a banking application or

in online sales. There is no compelling reason why all of these objects should be based on different models if a single model as a solution to all exists.

- Adaptability—much like applicability, adaptability refers to a software's ability to adapt to changes over space and time. For example, physical books may be replaced with e-books under certain circumstances. These are distinct objects that perform the same function but have different attributes. An adaptable system created in the absence of e-books will ideally still be able to handle their inclusion into its software despite the unanticipated nature of the innovation.
- Customization—there may be times in which the user will need a particular subset of a program's functionality or a subtle variation of it. Software that is customizable can meet the needs of these users.
- Personalization—a software program that is used in an unlimited number of instances will ultimately fail if it cannot be personalized to the end-user's needs. Any reusable software produced must handle the special needs and one-off cases of the client. This is done by making the object sufficiently abstract so that any user changes do not interfere with the normal operation of the software.
- Self-configurable—when the goals of a business change, the goals of its software should change with it. The software itself, however, should remain largely unchanged. The investment is too great to build a new system, and the old system is too unwieldy to alter except with extreme effort. Although small, invasive, and unpredictable changes can be made to an older system, it would be better for that system to reconfigure itself in order to meet the new objectives of the organization.

- Self-adaptive—things change with the times, and so too does software. Imagine the labor that could be saved if software kept up with the times on its own. This is the essence of self-adaptability. Such software can handle the creation of new objects and the release of older ones without any change in the structure of its program. This is accomplished by adequately mapping the knowledge of the purpose and function of all the components onto the software so that new items and ideas automatically fit into the logical framework.
- Self-managing—in modern software, there are many pieces of software that interact in sometimes unpredictable ways. These side-effects are known to cause many problems, and the solution has often been to isolate objects and prevent them from altering one another too much in an attempt to reduce unintended consequences. A better solution would be for the object to self-manage its accessibility with outside objects.
- Abstracted—abstraction is a key feature of modern software as it allows programmers to make generalizations that handle specific problems rather than considering all possibilities in advance, which is an inconceivably difficult undertaking. Though practically all software utilizes abstraction in some sense, the best software systems abstract only enough to make future modifications simple without abstracting so much that the system becomes unusable.
- Globalized—modern software must be designed with the global economy in mind. The market for international software is substantial, but software developed for that market must take a number of things into account, such as language and culture. As an example, the classic game of Minesweeper (available on any

Windows platform) was altered some time ago in certain locations where landmines are a legitimate danger. To avoid legal trouble and out of concern for the psychological well-being of its clientele, Microsoft fixed this by changing the art to show a flower garden instead. (Cobbett, 2009)

Overview of Software Reuse Knowledge Map (Core Knowledge)

EBTs represent the elements that remain stable internally and externally. BOs are the objects that remain internally adaptable but externally stable. The software reuse knowledge map consists of two EBTs (reuse and abstraction) and two BOs (assets and contexts). Each of these will be briefly discussed here before full patterns are given for each. For a comprehensive description of all EBTs and BOs associated with software reuse see Appendix A.

- Reuse means to use something again after it has already seen use. This includes conventional reuse in which the item is used again for the same function and creative reuse in which it is used for a different function. For example, while modular frameworks can be easily reused, test automation is often a very expensive (albeit valuable) effort and the Return on Investment (ROI) can become questionable. This is primarily because of changing product functionality that may invalidate the test scenario at hand. Although this challenge is often beyond the scope of the test team to control, the situation gets doubly complicated when poor test automation code is generated due to an equally poor and inexperienced choice of test cases. These tests can be very cumbersome to read through, review, and maintain. An answer to all this is to modularize the test automation code and create frameworks to handle repetitive functionality (e.g., code to log in or log out could be easily separated and handled in a separate module to be reused when required).

- Abstraction is the act or process of separation. It is the view of a problem that extracts information relevant to a purpose and ignores the rest. Abstraction is one of the convenient ways to deal with complexity.
- Asset can be defined as anything of material value or utility. There are several types of assets in USER based software reuse, such as architectural frameworks, architectural mechanisms, architectural decisions, and constraints applications.
- Context signifies the set of circumstances that surround a particular task undertaken. Software reuse depends on the context in which it is implemented and requires a systematic approach.

Asset Stable Analysis Pattern

An asset is a BO with the ultimate objective (EBT) of ownership. It is a reusable product or by-product of software development. Typical examples are code, design, specifications, user documentation, test plans, and estimates. Producers are the individuals or groups that create assets with the explicit purpose of reuse in mind. The users of these assets are consumers. When a producer makes an asset explicitly available for reuse by placing it in a reuse library, the asset is said to be published.

For example, producers successfully use the World Wide Web (WWW) to publish their assets, resulting in the creation of a gigantic, virtual database of reusable assets. The recent availability of WWW search engines has provided consumers a very powerful indexing mechanism with which to access this virtual database. In addition, standards for packaging assets have emerged on the Internet making reuse easier.

Assets are a resource owned by a specific party. The party specifies some criteria to describe assets. Assets have some types and each type has some entities or events. In addition, every asset has its evidence to prove the party's ownership, and the evidence is recorded in some log. The log, entity, and event are on some media.

Requirements. In the software reuse, assets have several requirements.

Non-functional requirements:

- Measurable—there is a specific quantity for an asset. This can often be the monetary value of the asset.
- Documentable—the asset must be recordable. For example, a bank account has a transaction history and inventory has a bill of sale.
- Usable—the asset can be used. For example, a business can spend money in a bank account, sell inventory, or rent building space.

Functional requirements:

- AnyParty—the asset is owned by a party, such as a person, a country or an organization.
- AnyCriteria—AnyParty has some criteria to evaluate the asset.
- AnyEvidence—when AnyParty claims it owns some asset, it needs evidence to demonstrate ownership.
- AnyLog—AnyEvidence needs to be recorded in some place such as a log.
- AnyType—the asset has many kinds.
- AnyEntity and AnyEvent—the asset can have many specific instances.
- AnyMedia—AnyLog needs media to store it.

Solution. The essence of an asset can be modeled and described as in Figure 3 below. Any asset, physical or otherwise, can be viewed as an extension of this pattern.

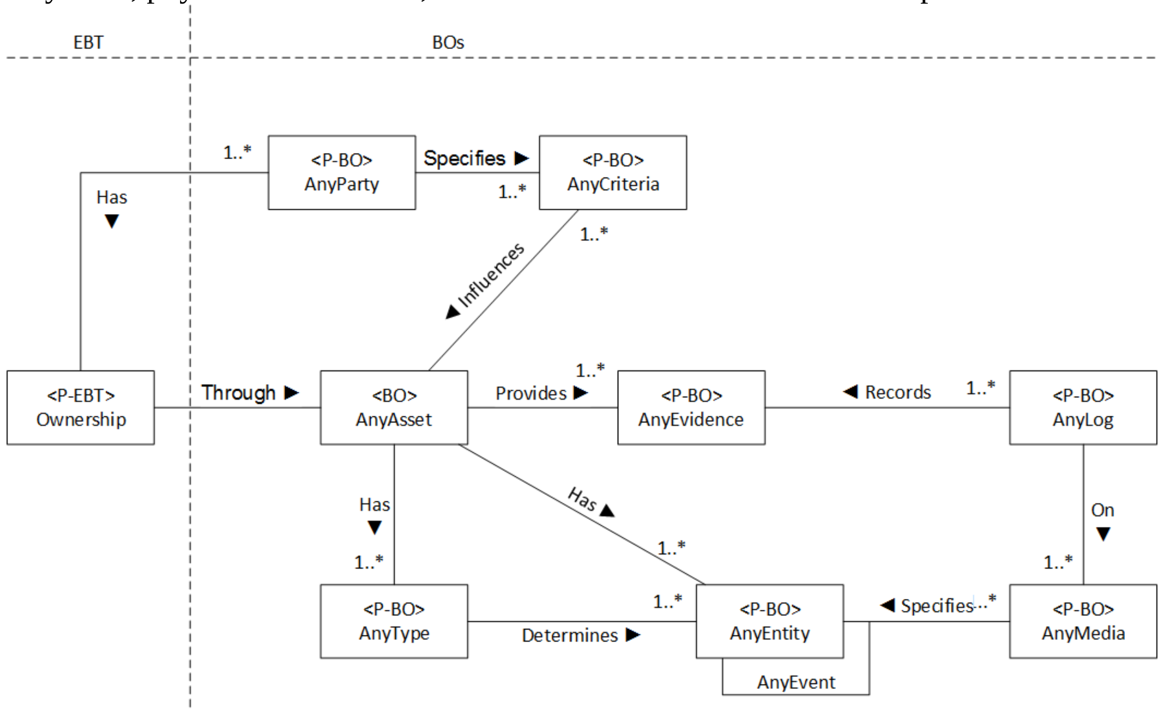

Figure 3. Asset stable analysis pattern.

- Ownership—the goal of any asset is ownership.
- AnyParty—the asset can only be owned by a party. Since ownership is dependent upon legal standing, AnyActor will never be a part of this pattern.
- AnyCriteria—AnyParty has some criteria for evaluating the asset.
- AnyEvidence—mere ownership of an asset is insufficient, especially when that ownership is contested between parties, so evidence of asset ownership resolves this issue.
- AnyLog—the place evidence is recorded.
- AnyType—the kind of asset.
- AnyEntity & AnyEvent—the other physical entities, events, or transactions.

- AnyMedia—the media for storing the log.

Application. Ownership can apply to a plethora of scenarios. Table 3 presents a variety of such scenarios to demonstrate this vast nature and the ability of the solution presented above to accurately model it.

Table 3.

Applications for Asset Pattern demonstrating ownership in various scenarios.

Pattern object	Banking	Real Estate	Intellectual property	Physical investments	Skills
AnyAsset	Bank account	House	Patent	Gold	Guitar skill
AnyParty	Bank, client	Lender, Owner	Government, Inventor	Owner, Insurer	Performer, audience
AnyCriteria	Bank policy	Value, Size, Comfort	Usefulness	Cost	Skill level
AnyType	Monetary	Physical	Intellectual	Physical	Intellectual
AnyEntity	Vault	House	Patent ID	Vault	Guitar
AnyEvidence	Receipt	Deed	Patent ID	Receipt	Recording
AnyLog	Transaction history	MLS	Paperwork	Receipt	Blog
AnyMedia	Paper, bank database	Paper, government database	Government database	Paper	The Internet

As an example of an application for ownership, consider the scenario of a bank account which is held in the bank. The client saves the deposit which the bank invests. The bank has specific policies by which the account is managed. When the client wants to prove or verify ownership of the account, the bank provides some receipts containing the account information to show the status of the account. All the monetary transactions can be found in the bank system. In this scenario, the bank and client are parties, the bank account is an asset, the bank policies are the criteria, the receipts are the evidence, and the transactions and bank system are the logs and media, respectively. The end result is a system much like the one in Figure 4.

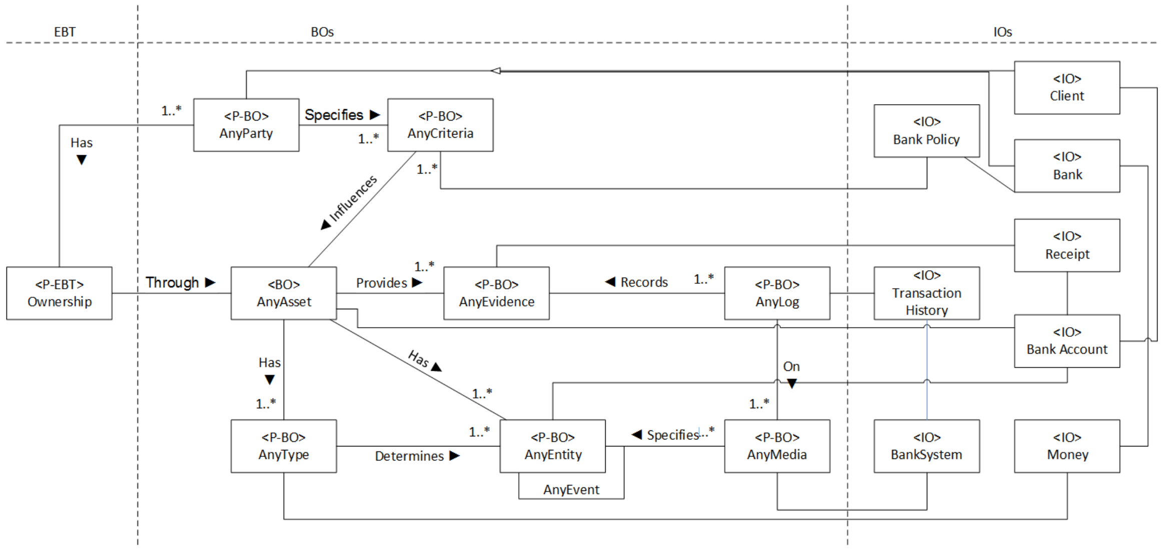

Figure 4. Asset application–a checking account.

In this application, one potential use case is that of a bank member depositing a check. A person is the party who plays the role of client. The bank is the party that plays the role of an organization. Table 4 further defines the use case classes.

Table 4

Use case classes for checking account (asset).

Class	Type	Attribute	Operation

Client	Person	id
		name
		gender
			requestAccountBalance()
			deposit()
			withdraw()

BankClerk	Person	id
		name
		gender
			printReciept()
			takeDeposit()
			interact(system,command)

SystemLog	System Class	id
		size
		data
			save()
			load()

Receipt	System Class	id name owner	print()

BankAccount	System Class	id
		amount
		type
			open()
			query()

BankPolicy	System Class	id name	checkCompliance()
Money	System Class	name id	payFor(item) Value
Deposit	System Class	id name create_time	deposit()

BankSystem	System Class	name
		platform
			update()
			acceptDeposit()

Use Case Description:

- The Client has Ownership of the BankAccount.
- The Client deposits a check with the BankClerk. (Does the client have a BankAccount? Does the client have proper ID?)

- The BankClerk interacts with the Bank System to add the Deposit to the BankAccount. (Is the BankClerk authorized to use the BankSystem?)
- The BankSystem checks the BankPolicy to ensure the Deposit is in compliance. (Is there sufficient Money to cover the Deposit? Is the check valid?)
- The SystemLog records the transaction.
- The BankSystem prints a Receipt. (Is there sufficient paper and ink to print?)
- The BankClerk hands off Receipt to the Client.

Reuse Stable Analysis Pattern

Reuse is an EBT, and its BO is Reusability. Software reusability is generally considered a way to solve the software development crisis. When solving a problem, it is best to try to apply the same solution to similar problems because that makes the work easy and simple. One thing is certain – software reusability should improve software productivity. Software reuse has become a topic of much interest in the software community due to its potential benefits, which include increased product quality, decreased product cost, and shorter schedule. The most substantial benefits derive from a product line approach, in which a common set of reusable software assets act as a base for subsequent similar products in a given functional domain. The upfront investments required for software reuse are considerable.

Requirements. There are several BOs that are a major part of the reuse pattern. These are some of the most prominent:

Functional requirements:

- AnyArtifact is a reusable unit absolutely essential for the software reuse.

- AnyMechanism is a natural or established process by which something takes place.
- AnyContext—defines the encapsulation in which one performs software reuse.

Non-functional requirements:

- Complete—software reuse has to be complete in nature. Its completion is determined by the fact that the design and framework of the software development support its reuse.
- Testable—software is testable if it supports acceptable criteria and evaluation of performance. The reuse must be able to be tested (e.g., one must be able to generate reusable test-cases for the reuse process). To reuse software cost-effectively, one must re-verify components in their new environment.
- Stable—it should be a stable process which remains for a definite period of time. One important question to consider is: "Are my software development environments, tools, and platforms well defined and stable?" If not, the developer should first focus on domain models and business requirements.

Solution. A banking scenario can serve to illustrate the solution for the reuse pattern. The money in the bank is an entity that, through the mechanism of a loan, can be reused for multiple transactions. The bank has criteria for loaning the money and always creates evidence in the form of a paper trail. More recently, this information is stored on hard drives of other digital media. Given this scenario, we see that reuse involves a mechanism, criteria, evidence, entities and media. This solution is displayed visually in Figure 5.

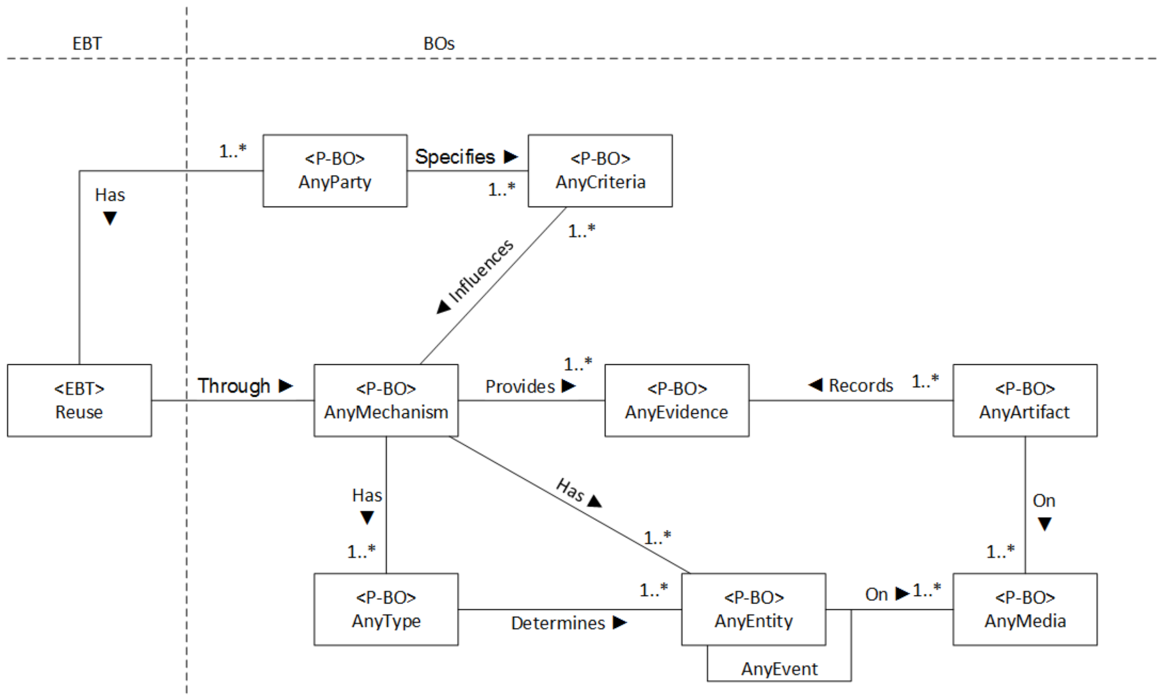

Figure 5. Reuse stable analysis pattern.

Applications. Reuse has many applications ranging from the recycling of physical substances, to the reuse of more ethereal things such as software. Table 5 shows a selection of several such scenarios.

Table 5

Applications of the reuse stable analysis pattern in various scenarios

Pattern object	Recycling cans	Water	Renovation	Compost	Software
Any Mechanism	Recycling	Recycling	Reuse	Composting	Reuse
AnyParty	Soda consumer, recycling center	Business, government	Contractor	Consumer	Programmer
AnyCriteria	Law	Law	Contract	Time	Time
Any Evidence	Money	Meter	Invoice		Document
AnyArtifact	Crushed cans	Water bill	Old wood deck	Compost	Software
AnyType	Material	Material	Material	Material	Information
AnyEntity	Aluminum	Water	Wood	Food scraps, organic material	Software
AnyEvent		Morning sprinkling	Deck removal		
AnyMedia	Cash, receipt	Water bill	Invoice		Hard drive

Abstraction Stable Design Pattern

The goal of abstraction is identification. It is the view of a problem that extracts information relevant to a purpose and ignores the rest. Abstraction is one way to deal with complexity. It plays a central role in software reuse. Concise and expressive abstractions are essential if software artifacts are to be effectively reused.

Requirements. There are several BOs that are required to support the abstraction stable design pattern, such as:

Functional requirements.

- AnyParty—any party, institution, organization, or individual can request the abstraction of a particular concept. This can also be any actor as abstraction can be found in software.
- AnyCriteria—an abstraction needs to be accompanied by the criteria for abstraction. Reuse, for example, could be one of the criteria for code abstraction.
- AnyContext—the context of abstraction helps define the criteria for the abstraction.

Non-functional requirements.

- Unique—the abstracted module needs to be unique to avoid any kind of redundancy in software. Redundancy could increase the development overhead.
- Definable—the abstraction must not be vague as this would defeat the purpose of identifying the scope and limitations of the program and its reuse.
- Accessible—the accessibility of a module is crucial for it to be reused anywhere else in the program.

Solution. When a party requests abstraction, the problem is classified and abstracted using a mechanism based on the context of any type of entities and events. This can be used to produce logs on any media. See Figure 6 for a visualization of the stable analysis pattern for Abstraction.

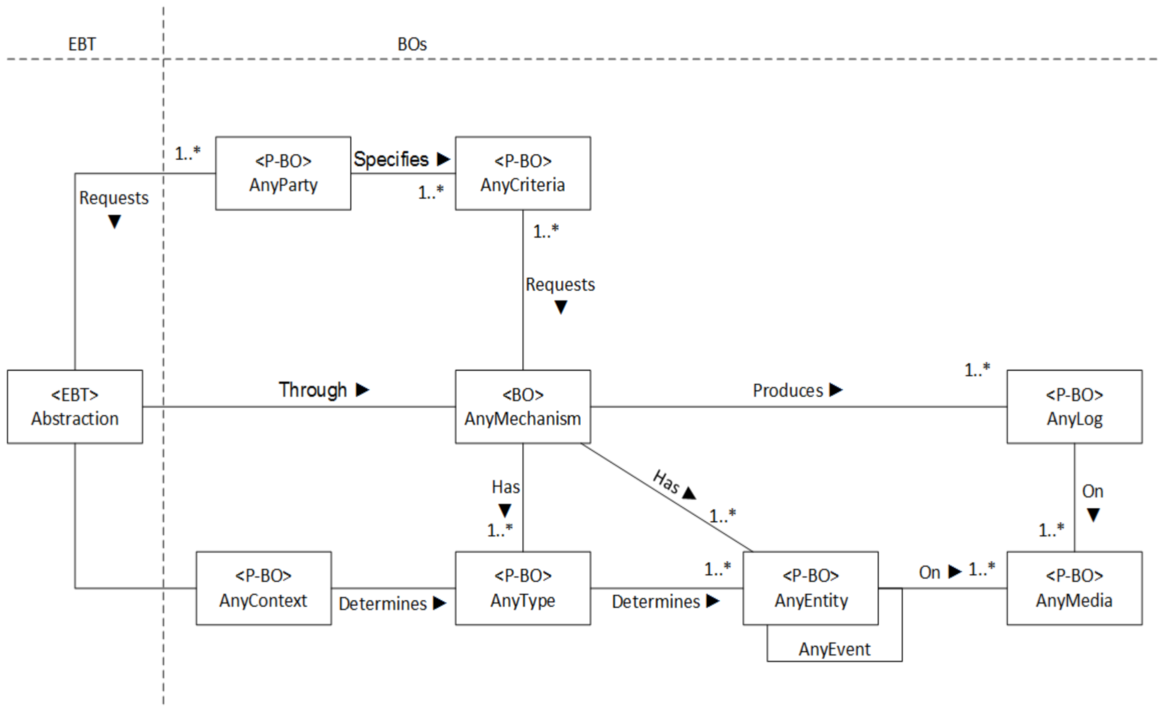

Figure 6. Abstraction stable analysis pattern.

Applications. Abstraction can be applied across many applications. Table 6 contains a list of several scenarios utilizing the concept of abstraction.

Table 6

Applications for abstraction stable analysis pattern in various scenarios

Pattern objects	Programming	Electronics	Vehicles	Language	Application
Criteria	Class module	Phone	Safety	Grammar, syntax	Mobile
Party	Developer	User	Driver, passenger	Communicator, audience	User
Context	Specific context	Mobile app	Driving	Newspaper article	Ticket booking
Mechanism	Programming	Operation	Self-drive	Operation	Online
Entity	Input	Person	Person	Word	Person
Log	Result	Picture	Destination	Dictionary	Ticket booked
Media	Computer	Camera	Hard-drive	Paper, ink	Internet
Type	Conceptual	Applied	Applied	Conceptual	Applied

Context Stable Design Pattern

The goal of context is encapsulation. The context of a system defines the relationship of the system with the environment. This includes the various constrains of the system that are at the organizational and program levels. The interaction and dependencies between these factors helps determine the scope of the project. Approaches that support software reuse and functionality can be adapted and configured for use in a specific context.

Requirements. The context design pattern, just as any other, has a number of supporting BOs and EBTs to fully model a given scenario or situation.

Functional requirements:

- AnyParty—any party, institution, organization, or individual can request the encapsulation of a particular concept.

- AnyCriteria—the encapsulation of a certain function would depend on criteria, such as if they are related to the same entity or similar operations.
- AnyScenario—encapsulation can be found in day-to-day activities. For example, using a single switch to turn on the lights as well as the exhaust fan in a bathroom.

Non-functional requirements:

- Specificity—the context of a problem should be specific in order to propose the right solution. In the same way, context defines the entire environment when encapsulated.
- Applicability—the encapsulation or wrapping of certain qualities into one entity should be applicable in the sense that the entities as whole should prove to be different from each.
- Reusable—the encapsulated functions of objects should be reusable by all the objects that fall under the same top level categorization.

Solution. When Context is involved, the solution can be estimated, but the context changes over time. Improving reusability means creating context-free software. See Figure 7 for a visualization of the stable analysis pattern for Context.

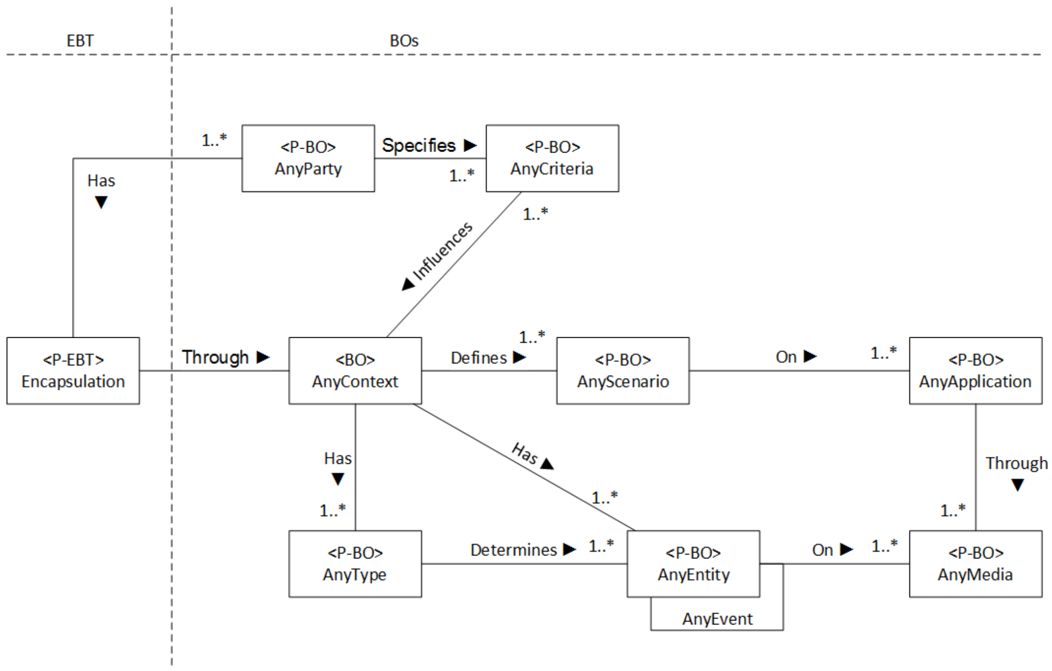

Figure 7. Context stable analysis pattern.

Applications. Encapsulation can be applied across many applications. See Table 7 for a breakdown of its use across security, debate, application, cooking, and education scenarios.

Table 7

Applications for context stable design pattern in various scenarios

Pattern object	Security	Debate	Application	Cooking	Education
Criteria	Safety	Ethos, pathos, logos	Light, Quick	Taste	Best education
Party	Security	Debater	Developer	Cook	Student
Context	Safety	Democracy	Traveling	Cooking	Software engineering
Mechanism	Checking	Speech	Programming	Baking	University
Entity	Person	Person	Input	Person	Person
Log	Registry	Video	Result	Cookbook	Student database
Type	Screening	Interactive	Mobile App	Food	Classroom

Summary

Achieving software reuse is extremely challenging. Large scale, systematic reuse is more difficult in an organization. Developers with deadlines to meet and functionality to deliver may find it challenging to keep reuse a priority.

Software reuse, though not particularly convenient or easy to achieve initially, is a viable concept that is well worth the effort. In business, lack of leadership and vision concerning software reuse within an organization's political and cultural context is a key factor. Some efforts fail because they are overly ambitious and many large upfront design efforts are spent trying to design things future-perfect. Non-alignment with what business clients desire to accomplish also creates problems for reuse minded developers. Still others fail due to lack of design flexibility, inadequate planning, and funding issues. Finally, communication effectiveness and awareness of existing reusable software assets are additional critical factors. Despite all of these troubles, however, true software reuse is a tantalizing and attainable goal.

Chapter 4: Context Based Software Development Knowledge Map

Introduction

The elements existing across the development process that define the ultimate goal of the developers' team is their context. This context is essential to define for software developers and engineers to easily recognize the boundaries and scope of the software under development. This chapter clarifies the essential elements and properties of context based software from the perspective of SSM. It also illustrates the knowledge maps for a few of those elements and properties in the form of stable design and analysis patterns. With a firmer understanding of the constituent components, the context based software development knowledge base can be advanced and used more effectively

In software engineering, context usually refers to the environment in which a piece of software is placed, potentially influencing its operation (Tamai & Monpratarnchai, 2014). Writing software in such a manner is theoretically designed to increase the adaptability of the software in a given context by dynamically binding objects to roles. However, the adaptability of this system applies only to the scenario specifically developed, and then, only in the time and place it is developed is it certain to function as expected. This level of system adaptability falls far short of what is possible; systems should be able to be easily adapted to new scenarios to maximize reusability and reduce maintenance. If context oriented software is to be made so adaptable, a new, more expansive definition of context is needed.

In fact, it is actually difficult to derive such a general definition or meaning of context. Any context should be very well connected or related to the entity, should lead to the solution of the problem, should evolve over time, should have a dynamic process in a very dynamic environment, and should relate to the domain of use and time.

The features mentioned will give different problem statements and pose a number of challenges to software developers. Due to these features, it is also difficult for software developers to have just one dimension and context. Software development projects easily become humongous, incredibly complex, and hard to capture as whole. During the development phase, it is difficult for software developers to grasp the context of software development. They have to read and understand a lot of contextual information that is typically not processed and captured to improve their working environment.

The aim of this chapter is to illustrate context in the software development domain with the help of knowledge maps. These knowledge maps are explained with short and mid-sized templates of applicability, unification, and scenario. In addition, the work is intended to give a new perspective to developers who are working toward the expansion of context driven software development.

The Essential Elements of Context Based Software Development

Context based software development involves a number of important elements:

- Application in context based software development refers to the use of the context model and the ultimate goals, aims, and objectives behind its use.
- Patterns are available in two types: dotted and filled patterns. Dotted patterns represent explicit relations, whereas filled patterns represent implicit relations. In addition, common patterns are used to perform search operations (Riehle & Züllighoven, 1996).

- Scenario is where most of the actions take place. Analysis of various scenarios can be used to support the work of a developer by providing additional knowledge and enhancing the quality of the work environment.
- Context represents a complex network of elements across various dimensions that are not limited to the work developed in an integrated development environment (IDE). Context usually takes into account all of the dimensions that characterize the work environment of the developer.
- Design must be executed in such a way that various dimensions can be represented as a layered model. This model consists of four main layers: personal, project, organization, and domain.
- Architecture for the context model is layered. Therefore, by each layer of the proposed context, the model will define which context to capture, the type of modeling, its representation, and the kind of application for that layer.

Given the elements mentioned above, context-driven development follows a natural progression from the context to the application. The context is used to describe a scenario that can then be abstracted into a pattern. This pattern is used to create a design which subsequently yields an architecture for the application, as shown in Figure 8. For a full list of all EBTs and BOs used in the context based software development knowledge map see Appendix B.

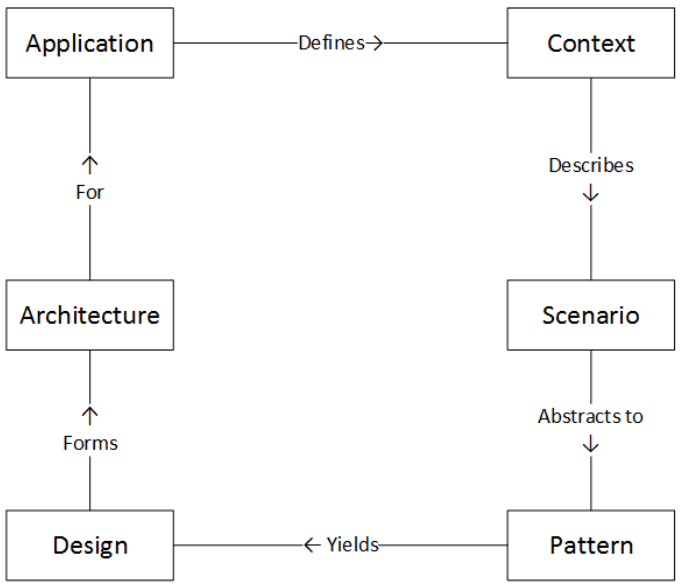

Figure 8. Elements of context based software development.

Context Based Software Development – Essential Properties

There are a number of essential properties implicit in context based software development:

- Simplicity—organizations or people (AnyParty, AnyActor) involved in software development need to have their components (AnyEntity) chosen based on certain conditions (AnyCriteria) for the ultimate result (AnyOutcome).
- Adaptability—the software development team (AnyParty) or the software components (AnyEntity) need to have an ability to change something (through AnyMechanism) to cope with random unexpected changes (AnyEvent, AnyImpact).
- Extensibility—all the remaining layers (AnyEntity) of the context model other than the original layers (Personal, Project, Organization, Domain, Any Party) will

be referenced (AnyEvent) repetitively (through AnyMechanism) as an extension (AnyOutcome, AnyLevel) of the already developed work (AnyCriteria).

- Customization—information retrieval facilities needs to be customized (AnyCriteria) by the software developers (AnyParty) to facilitate the support (AnyEvent) of reusing (through AnyMechanism) software components (AnyEntity).
- Abstraction—all the dimensions or layers (AnyEntity) modeled in context based software development are to be abstracted or hidden (AnyEvent) from the implementation by the development team (AnyParty) and cannot be seen (AnyImpact, AnyOutcome) by a user (AnyActor) directly.
- Applicability—the organization or a software development team (AnyParty) requests applicability, which represents (AnyEvent) how the context model (AnyEntity) is used (AnyEvent) and the objectives behind its use.
- Unification—all dimensions (AnyEntity) of context information (AnyContext) provided by the working environment of a developer (AnyActor) need to be integrated (AnyCriteria through AnyMechanism) for creating a unified context model (AnyOutcome).
- Reusability—software components (AnyEntity) need (AnyCriteria) to be reused (AnyEvent) by focusing on the information captured (AnyOutcome) during the process development (AnyMechanism).

- Configuration—software configuration (AnyOutcome) is the task (AnyEvent) of tracking changes in software (AnyEntity) and controlling them (through AnyMechanism).
- Personalization—a developer (AnyActor) working on a specific task (AnyEvent) needs to deal with various resources (AnyCriteria) for accomplishing the task (AnyOutcome).

Though each of these properties is important to context based software development, a greater focus on applicability and unification will provide a better understanding of the nature of these properties.

Applicability Stable Analysis Pattern

Applicability becomes a tool to develop a concrete pattern. It forms a model in any scenario wherein one finds applicability in context based software development. Figure 9 includes a stable and robust design pattern for applicability in context based software development. It applies to all the scenarios of a context based software development process.

Applicability can be generally defined as the utility of something for a particular task. For example, books have great applicability for learning and gaining knowledge. In other words, they are suitable and useful when accomplishing a task of learning. A book or a journal has more applicability in a library. Likewise, a drilling tool is useful for a carpenter or a mechanic, but it cannot be used other than for the tasks that needs it. Hence, the context in which something is used is very critical for greater applicability and effectiveness. In the domain of stable pattern development, the term context denotes meaningful applicability. Applicability is the main business theme of a context based stable pattern.

Requirements. A number of BOs and EBTs are required to satisfy the many scenarios to which applicability man apply. The most vital are presented here.

Functional requirements:

- AnyActor—the person who is a part of the system and interacts with the system is known as AnyActor. AnyActor can be a creature, hardware, software, or any person that acts. In this context, it can be a person, a startup, or an organization that develops context based software.
- AnyParty—a party is any organization, political party, or group of individuals having similar ideology and concepts. In this context, an organization that develops context-based software is a party
- AnyCriteria—a concept or reason on which one makes a judgment or decision is known as AnyCriteria. A set of criteria must be defined before developing a context based software. Requirements of the actors, such as special business rules and unique features, that they want in software is AnyCriteria.
- AnyRule—rules, regulations, and policies imposed by a party or actor are AnyRule. For example, when a customer has a unique condition from the organization that the cost of the software should not exceed a particular amount and it should be delivered according to a certain schedule, then these are AnyRules.

- AnyMechanism—the process used to achieve the desired goals is a mechanism. For example, when adopting new methods and design patterns in context based software development or when trying to have a new approach to the development process, it can be termed as AnyMechanism.
- AnyStage—the duration or period in which a particular step occurs or activities of context based software development happen. For the development of context-based software, one may have various activities like identifying the right resources, planning the development task, dividing the tasks among different teams, following a certain timeline to achieve desired outcome. These steps should be executed in a series.
- AnyDuration—the period or the time it takes to develop context-based software is known as duration. There are certain set deliverables according to the timeline.
- AnyType—this is a class or group which can be identified in context based software development. In this case, the context used for software development forms a legitimate example of AnyType.
- AnyEntity—these are the entities and attributes used in context based software development.

Non-functional requirements:

- Cost effective—context based software development should be cost effective and within the actor's budget.

- Creative—context based software development should be creative. It should always set a high standard by its creativity and in the final product or the approach taken to build it.
- High quality—the factors that set the final product apart from its counterparts form the software's quality factor. In terms of context-based development, the software should not crash and should be very responsive.
- Feature rich—context based software should contain many features that meet the needs of the client.

Solution. A model for applicability can be constructed based on the requirements presented above. This applicability stable analysis pattern is described in Figure 9.

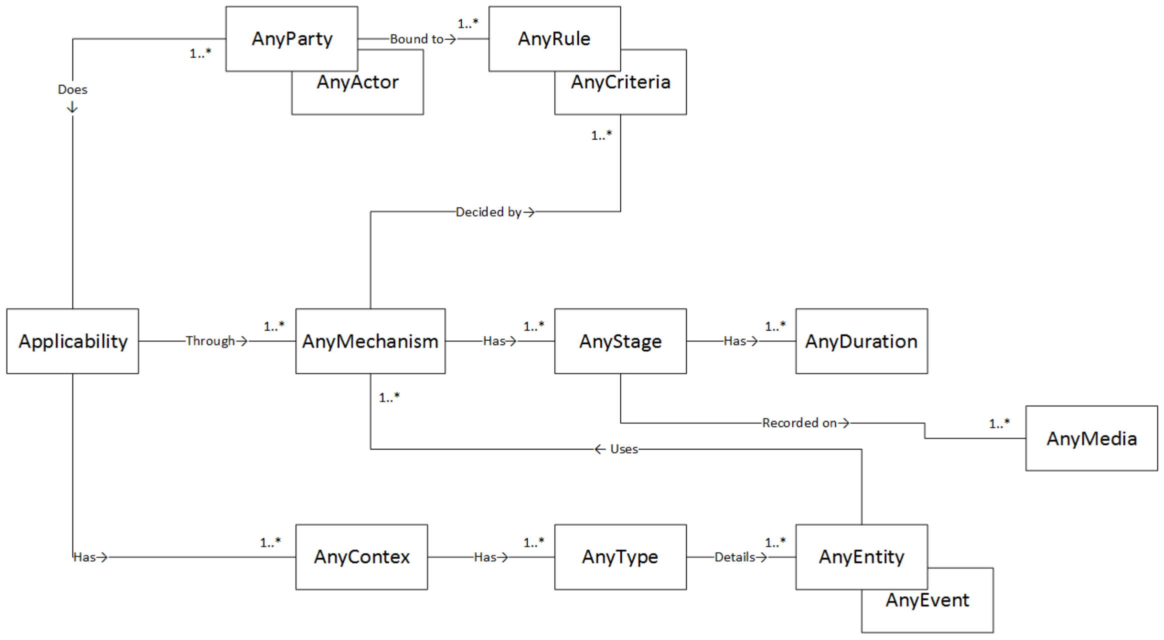

Figure 9. Applicability stable analysis pattern.

The solution to the problem is found in the applicability stable analysis pattern:

- The applicability of context based software development is done by AnyParty.
- Applicability is done through AnyMechanism.

- AnyCriteria or AnyRule should bind AnyParty or AnyActor.
- Any Mechanism depends on or is influenced by AnyRule or AnyCriteria.
- AnyMechanism has a duration and a series of steps associated with it.
- All the steps in the process have a stipulated duration or timeline in which they are to be completed.
- Applicability has a particular or specific context.
- Each context also has a type of software associated with it.
- Context based software development is applicable to and is developed for a specific entity.

Applicability. The concept of applicability can itself be applied to a number of situations. Some potential applications can be found in Table 8.

Table 8

Applications of applicability stable analysis pattern.

Pattern objects	Personal software	Company's internal tool	Web-based tool	Native tool	Third party tool
AnyParty	Student, person	Company	Company	Company	Company
AnyCriteria	Personal project	Internal company project	Scalable project	Platform-specific	Web service
Any Mechanism	Professional development	Employee productivity	Available anytime, anywhere	100% utility	Ease of use and integration
Any Duration	Month	Month	Month	Week	Month
AnyType	Term Project	App	Resource planning	Android app	Outsourced tools
AnyEntity	Person	Company	Company	Company	Company
AnyRule	Useful, creative	Cost effective	Cost effective	Cost effective	Replaceable, modular
AnyStage	Planning, design, execution	Planning, design, coding, testing	Planning, design, coding, testing	Planning, design, coding, testing	Planning, design, coding, testing
AnyMedia	Hard drive	Hard drive	Remote server	SD card	Hard drive

Scenario Stable Design Pattern

The stable design pattern for Scenario is a pattern that describes a situation for gathering requirements. Scenario works on any software component for context based software development. Scenarios, such as collaboration, integration, and design, can be achieved by devising specific scenarios.

Requirements. Scenarios can be used in wide variety of applications, but the problem is arriving at a stable design pattern for it. Presented here is a stable design pattern, which can be applied across various domains that involve a scenario. The scenario design pattern is needed for a system that deals with different situations and needs to be flexible in every situation.

Functional requirements:

- AnyParty—AnyParty refers to a person or a team who prefers to participate in the scenario.
- AnyScenario—AnyScenario is the situation that needs to be set up for an operation to take place.
- AnyEvent—AnyEvent takes place in a system while performing an operation.
- AnyContext—AnyContext is the context of the scenario for a particular operation.
- AnyDuration—AnyDuration is the duration for which an event takes place.
- AnyMedia—AnyMedia is an environment in which the operation takes place.
- AnyEntity—AnyEntity is considered with the type of props used for a scenario like integration, design, and collaboration

Non-functional requirements:

- Software synchronization—Software synchronization is the process of making different components of software or different software applications work together at the same rate or at the same time. It can be achieved under a scenario, which can be set by any party and according to any context.

- Software evaluation—Software evaluation in a context based technology environment is a process to discover the exact fit between the technology and the system. It can be achieved by AnyParty under AnyScenario for AnyContext and in AnyMedia.
- Software robustness—Robustness is the property of withstanding a load. To withstand a heavy load, developers set up a scenario in which AnyParty can perform load tests according to AnyContext and for AnyDuration. It can be done using AnyMedia and produce AnyLog.

Solution. Based on the requirements listed above, it is possible to construct a model for a generic scenario. The resulting class diagram of the pattern is shown in Figure 10. To show the potential range of functionality, various applications for the scenario pattern are given and described in Table 9.

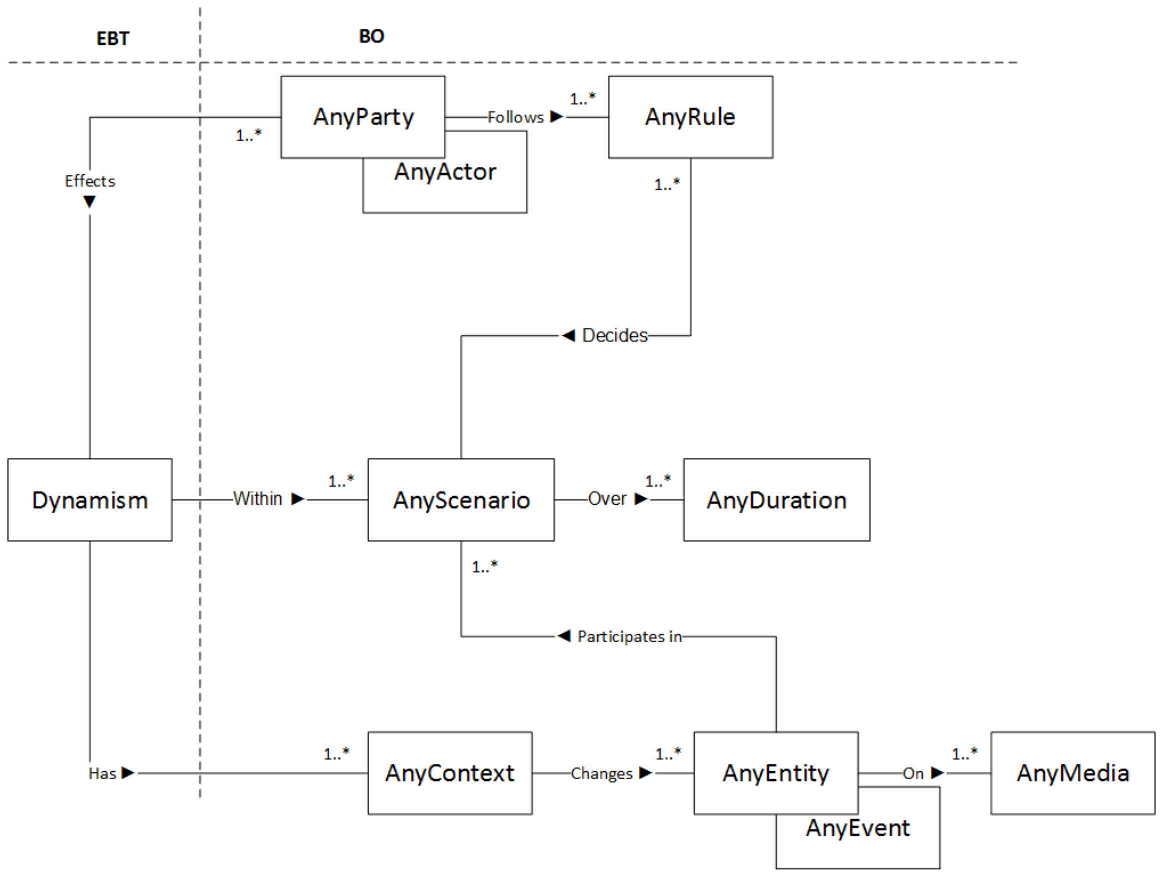

Figure 10. Scenario stable design pattern.

- Dynamism is the EBT for Scenario.
- Each party specifies dynamic changes.
- AnyParty follows AnyRule.
- Dynamism works according to a scenario and has context.
- AnyContext has an entity and an event.
- AnyContext changes AnyEntity and happens in AnyMedia.
- Dynamism—the EBT for AnyScenario.
- AnyParty—any person or group of people who bring about dynamism.
- AnyRules—the rules specified by AnyParty.

- AnyScenario—the situation for which the party specifies dynamism.
- AnyDuration—the length of time for which dynamism occurs.
- AnyEntity—the entity that is being changed.
- AnyEvent— takes place when a change is made.
- AnyMedia—the place or media where the event occurs.

Applicability. Scenarios can be used in many unique situations. To show the potential range of functionality, various applications for the scenario pattern are given and described in Table 9.

Table 9

Applications of scenario stable design pattern in various scenarios

Pattern object	Medical	War Game	Business	Software	School
AnyParty/ AnyActor	Doctor, patient	Commander, soldiers	Company	Developers, testers, marketers	Teacher, student
AnyRule	Biology	Military objective	Economics	Customer requirements	Grade scale
AnyScenario	Child diabetes	War game	Hostile takeover	Software development	Midterm exams
AnyDuration	Years	3 Days	9 Months	6 Months	1 Week
AnyEntity	Disease	Civilians, assets	Brand, employees, product	Servers, software	Exam, course content
AnyEvent	Spread, surround	Engagement	Buyout, budget cuts	Milestone achieved	
AnyMedia	Patient chart, prescription	Hard drive, reports	Fiscal reports	Hard drive	Report card

Unification Analysis Design Pattern

Unification is a generalized pattern that can become a concrete pattern to form a model in any scenario.

Context. Unification stable design patterns can be applied to any domain in which a group of entities works toward achieving a particular goal. Hence, a generic entity works for a given duration to develop and improve the quality of context-based software. Reasons and criteria are the dominating factor in unification. This can come from taking a piece of software and developing a completely new design to cater to the needs of the end customer.

Requirements. There are many BOs and EBTs required to make use of Unification. The most important are described here.

Functional Requirements:

- Any Actor—the person who is a part of the system and interacts with the system. It can be a creature, hardware, software, or any human. In this context, an actor may be a person, a startup, or an organization that develops context based software.
- AnyParty—any organization, political party, or group of individuals having similar ideology and concepts. In this context, an organization that develops context-based software is a party
- AnyCriteria—a concept or reason on which one makes a judgment or decision. A set of criteria must be determined before developing context based software. There are certain requirements given by the actors, such as special business rules and unique features that can be termed as AnyCriteria.

- AnyRule—a set of rules, regulations, and policies imposed by a party or actor. For Example, when a customer has a demand for the organization that the cost of the software should not exceed a particular amount, it should be delivered according to their schedule, then it can be termed as a rule.
- AnyMechanism—the process used to achieve the desired objectives. For example, when one is trying to adopt new methods and design patterns in context based software development or when he is trying to create a new approach to the development process, then it is AnyMechanism.
- AnyStage—the duration or period in which a particular step or certain activities occur. For the development of context based software, there are various activities such as identifying the right resources, planning the development task, dividing the tasks among different teams, and following a certain timeline to achieve desired outcome. These steps should be executed in a series. This can be termed as any stage.
- AnyDuration—period or the amount of time it takes to develop context-based software. Certain deliverables are spread across the timeline.
- AnyType—class or group which can be identified in context-based software development. In this case, the context used for software development forms a legitimate example of AnyType.
- AnyEntity—the entities and attributes used in context based software development.

Non-functional Requirements:

- Iterative—unification carried out in context based software development should be iterative. Many rounds of iteration should be carried out while introducing unifications of context based software to achieve the desired output and optimum performance.
- Linear—unification should occur in a linear fashion. When a group of entities focus on business modeling, the next set of entities should focus primarily on requirements. This way, the team working on unification steadily learns about the problem before learning about the solution.
- Sequential—use cases of context based software development evolve through the core discipline during every round of iteration. The team carrying out unification learns more about the solution of a limited portion of the problem as the effort progresses across various phases. This results in a system that addresses a subset of the requirements, which may or may not be deployable or usable throughout the development cycle. The result of unification is a complete working system.
- Balanced—unification in context based software development should always be balanced. When applying unification to any type of context based software development, there are various criteria to be followed in order to make the complete system well balanced after unification is done. The roles, activities, and artifacts that address the risk and enable project success are the most common nonfunctional requirements when performing unification of any context based software development.

Solution. Figure 11 includes the stable design pattern for unification in context based software development. It is applicable to all the scenarios of the context-based software development process.

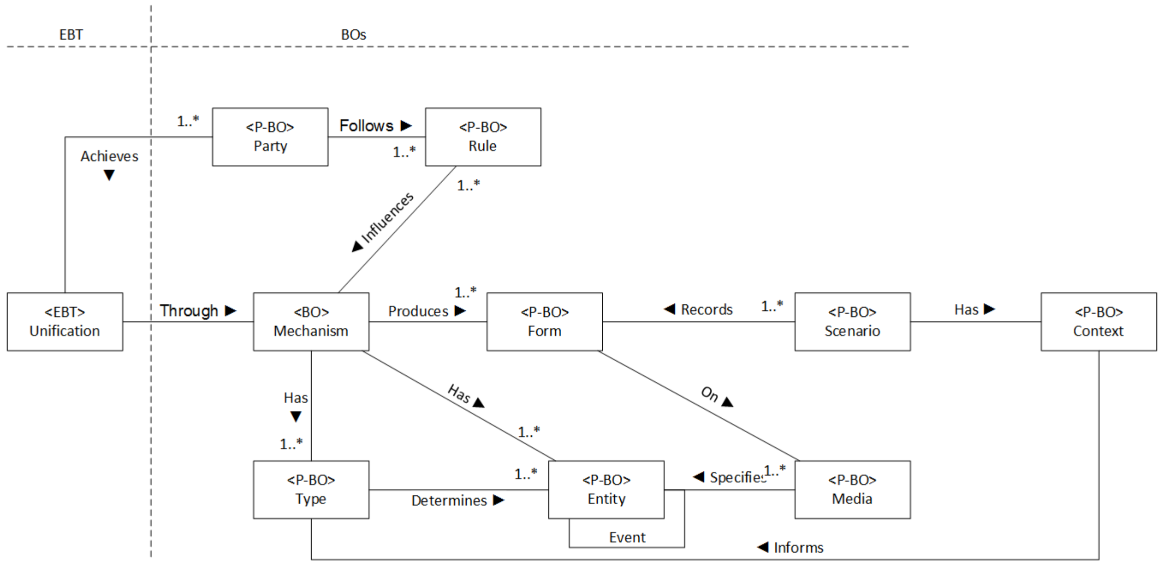

Figure 11. Unification stable analysis pattern.

- One or more parties or actors follows one or more rules to achieve unification.
- Unification is done through AnyMechanism.
- The rules influence the mechanism.
- AnyMechanism has AnyType which determines the entities or events involved in the mechanism.
- AnyMechanism produces AnyForm, based on the scenario.
- The scenario has a context depending on the type of mechanism.
- AnyForm is stored on AnyMedia.
- Unification is an EBT for context based software development.

- Any Party refers to the group of people involved in the unification process of context based software development.
- AnyCriteria is the set of conditions specified by AnyParty.
- AnyMechanism is the steps involved in the unification process.
- AnyReason explains the basis for unification in a context based software development process.
- AnyEntity is the type of context based software development that is subject to unification
- AnyEvent refers to the steps that are triggered for AnyDuration during the unification process.
- AnyType is the type of context based software that is being unified.

Applicability. Unification can apply to a multitude of situations. A diverse selection of these potential scenarios is described in Table10.

Table 10.

Applications of unification stable analysis pattern.

Pattern Object	Mathematics	Business	Political	Military	Marriage
AnyParty/ AnyActor	Computer science student	Microsoft, LinkedIn	Republicans, democrats	U.S., U.K., France, Germany, etc...	State, bride, groom, church
Any Context	Homework problem	Corporate Acquisition	Post-election atmosphere	NATO	Traditional marriage
Any Mechanism	Algorithm	Acquisition	Transition of power	Defense treaty	Matrimony
AnyEntity	Equation	Employees	Government	Military bases, troops	Certificate
Any Scenario	Class homework	Microsoft acquires LinkedIn	Citizens accept elected leader	Countries seek defense against a common foe	A man and woman marry
AnyMedia	Paper	Paper		Paper	Paper
AnyForm	Homework	Contract		Treaty	Marriage license
AnyType	Math	Business	Political	Military, political	Personal
AnyLevel	First-order		Spiritual	International	Personal
AnyEvent	Algorithm start	Negotiation	Election	Training exercises	Wedding

Conclusion

This chapter presented one approach for context based software development. The context model discussed here was based on a layered structure that considered four major dimensions or layers of the work environment for a software developer (personal, project, organization, and domain). Keep in mind, each layer should be defined by taking into consideration capture, modeling, representation, and application. The document defines a knowledge map of various recognized EBTs and their respective business objects in the context based model of software development. Various common patterns were mapped across all the EBTs. The network that exists between developers, different tasks, and resources are represented in the context model of the developer.

Chapter 5: A Pattern for Stress and Resolution

Introduction

When discussing stress, it is useful to specify the certain terms that are likely to emerge in the discussion, such as subject, stressor, coping, and stress. For the purposes of this document, stress will be a potential harm that can cause physical harm without itself being physical. Since stress must be considered in its relationship to an involuntary individual, the stressed individual will herein be called the subject. Stress will be considered as originating from a source external to the subject, called the stressor. The subject's method for dealing with the effects of stress will be called his or her coping mechanism. These are the elements common to all patterns that will be explored here.

This chapter will first explain some of the existing traditional models for stress. A similar description and explanation for a model generated using SSM (SSM; Fayad & Altman, 2001) will be added. This will be followed by a criterion-based comparison of the models, and, finally, an analysis of the applicability and implications of the new model discussed here.

Traditional Stress Model

There are a number of existing models for stress, each ranging in complexity and applicability. Most models cover only certain aspects of life, such as workplace stress. These have many competing models and theories, such as transactional, person-environment fit, conservation of resources, and demand-control (Dewe, O'Driscoll, & Cooper, 2012). While important, each of these models is somewhat lacking in applicability because stress on any aspect of life is stress on the entirety of a life. An example of workplace stress in the traditional style is shown in Figure 12.

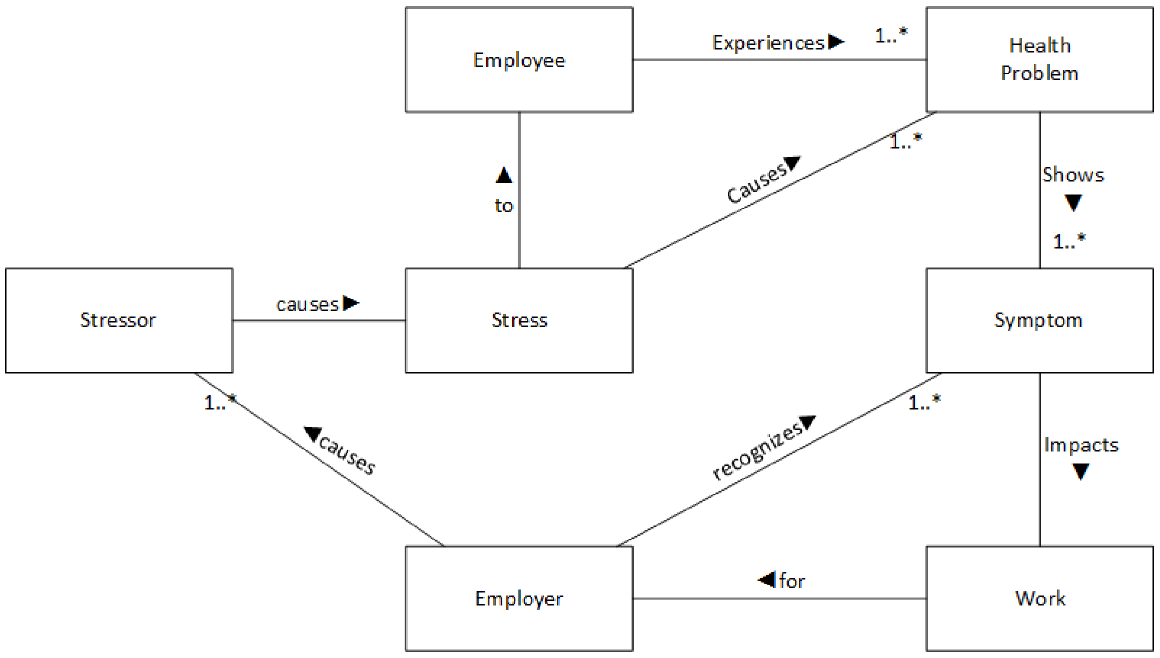

Figure 12. An example of a tradition model for workplace stress.

This particular model operates under a number of assumptions. First, the model begins with any number of stressors that causes the perception of a threat, real or otherwise. Dependent on the subject's reaction to these perceived threats, the potential exists for the subject to experience a duality of stress and fatigue which subsequently must be handled. If the subject is unable to cope with the stress or in some way mitigate its effects, the result is damage to the subject's physical and mental well-being.

An example of this pattern is moving a deadline at work, which becomes the stressor. When the deadline moves closer, an employee (the subject), may see a threat to his or her job security if he or she believes he cannot finish in time. Over time, stress causes health problems if it remains unresolved. The subsequent symptoms impact work performance, which the employer is likely to notice. The employer may react to the situation by limiting the stressor or firing the employee.

Stability Software Modelling

SSM (Fayad, private communication October, 2014) for stress, as presented in Figure 13, has a number of features unique to the stable software methodology. The central component of the model is stress which is merely the result of pressure. The pressure is caused by a reason that subsequently determines the type of the pressure and resulting stress. There must also be an actor or party, namely the subject that has the pressure and may feel stress based upon criteria. Though the subject may be a party or a legal entity, it is more likely to be a single actor in this case, such as a person or an animal. Finally, stress is manifested by evidence and results in damage to entities or events connected to the subject.

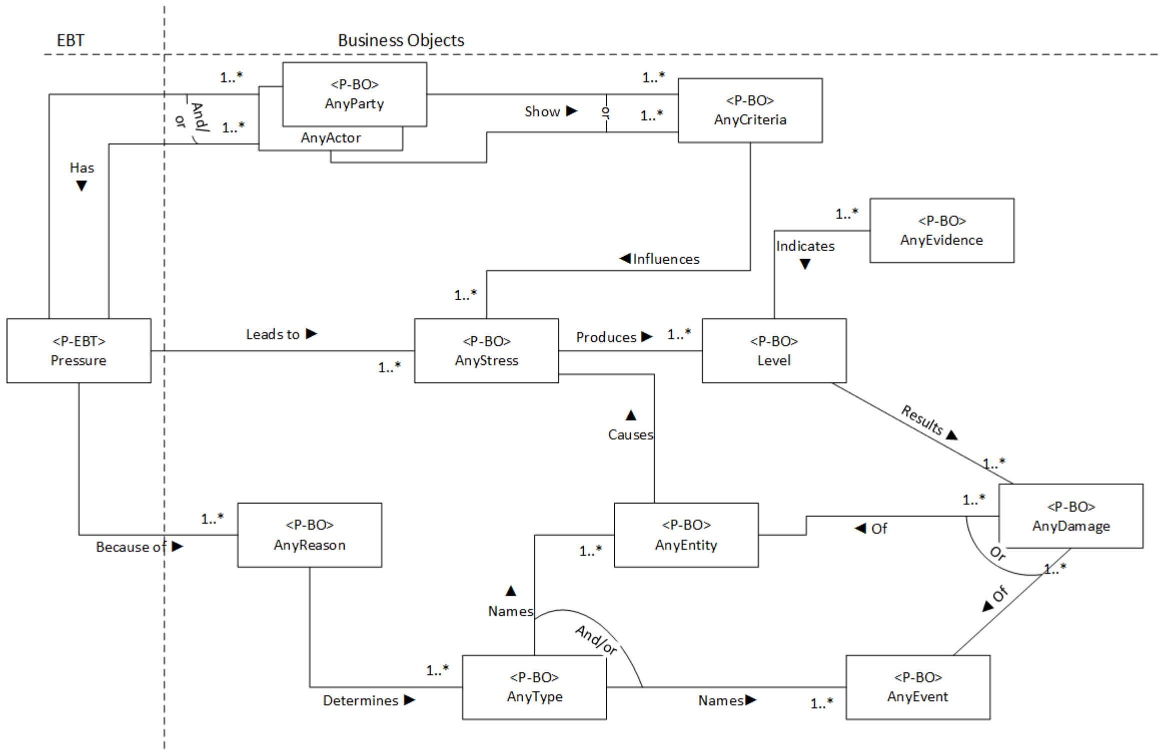

Figure 13. A stable model for stress.

Of special interest is the manner in which the stress may be resolved. If the reasons for the stress cease, the pressure and stress are reduced. More importantly, if the subject chooses to alter his or her criteria and become more or less accepting of the situation that reduces the stress level.

Consider the subject mentioned in the previous example whose deadline moved closer and who spent more time at work instead of at home with family as a result. The subject, the actor or party in the model above, experiences pressure from both work and home due to the long work hours, the reason for the stress. If the subject deems the reason serious enough based on personal criteria, stress will take effect. Lower quality work and neglect of family are the evidence that reflect the level of the stress If the subject cannot prevail upon the boss to readjust the deadline, the only option short of quitting the job is to change his or her personal outlook on the situation, the personal criteria, to minimize the overall damage.

Weighted Comparison of Traditional and SSMs

Both the traditional and stable models shown above may be compared based on certain criteria. A total score of 100 points will be split among these simple criteria.

- 20 points for simplicity—the model scores high if it is not graphically complex;
- 20 points for completeness—the model scores high if it is accurate for given scenarios;
- 20 points for stability—the model scores high if it is stable across multiple contexts;
- 20 points for clarity—the model scores high if it is easily understandable to a layperson;
- 10 points for testability—the model scores high if it can be readily tested;

- 10 points for extensibility—the model scores high if it allows for adaptation through extension.

With these criteria, it is possible to compare the stable stress pattern with other conventional models of stress. The results of the analysis of both models are summarized in Table 11.

Table 11.

Comparison of traditional and stable stress models.

Criteria	Weight	Traditional Model	Score	Stable Model	Score
Simplicity	20	The traditional model has few classes and minimal connections. It is about as graphically simple as a model of any substance can be.	20	The stable model has several classes and a low to moderate level of interconnectivity. The simplicity is commensurate with the complexity it attempts to envelope.	18
Completeness	20	The traditional model is only partially complete. It lacks support for actions taken by the employee, both to mitigate the stress and the resultant health problems. It also fails to address non-medical ramifications of stress.	5	The stable model covers the entirety of the scenario presented in workplace stress. Employee actions and subsequent ramifications are addressed.	20
Stability	20	The traditional model presented here is applicable only for the given context. This model is incapable of handling stress in any non-corporate scenario.	0	The stable model is applicable across all scenarios of stress in an individual's life, regardless of the context. Additionally, this model also applies to stress in engineering contexts as force on materials rather than weights on the mind of an individual.	20

Criteria	Weight	Traditional Model	Score	Stable Model	Score
Clarity	20	The traditional model, due in part to simplicity, is relatively simple to understand and difficult to misinterpret, even for those with little to no understanding of UML.	20	The stable model, due to increased complexity, lack in clarity somewhat. Experience reading stability models or a written explanation are useful for grasping the initial concepts.	17
Testability	10	The traditional model is small meaning there are fewer tests required. However, the connections between classes have subtle nuances that could be missed in the tests leaving gaps in the test coverage.	5	The stable model is of greater complexity and will thus require more tests. However, the function of each class is specifically spelled out, meaning that test coverage will be more complete.	5
Extensibility	10	The traditional model could, in theory, be extended to cover stress impacts that are non-medical in nature, but this would be difficult and the model would still be tied to the context.	2	The stable model is infinitely extendable, primarily in its capacity to apply to multiple scenarios and contexts. In any such situation, a new object is merely attached to the central described here.	10
Total			52		90

Discussions and Analysis

Abstraction. Given the weighted scores, the stable software model for stress is more complete and applicable to a far wider array of scenarios than the traditional model. The stability model makes sacrifices in clarity, since it is understood primarily by those who are familiar with Unified Modeling Language (UML). Previous experience with stable modeling is an additional help, but the model is sufficiently simple to be understood with only a small measure of explanation. In contrast, the traditional model is relatively simple to understand, but it may not be accurate and certainly will not address problems outside of its extremely narrow scope.

Application. The stable model has an advantage in both accuracy and flexibility. It can be applied to stress from any source for any reason with any impact as opposed to just stress originating from work and resulting in fatigue and physical symptoms. Several engineering fields talk of stress in scenarios outside of the workplace such as wind stress or torsion stress on physical, constructed objects. Not only is the traditional model shown here unable to handle domestic stress, but also it neglects stress that is outside the scope of human emotions. Stress in an outside environment and in human emotions is only addressed simultaneously by stable models.

Impacts. The stable model offers some measure of hope that stress can be handled with more than just coping skills. While it allows for the subject to cope with stress, there are additional options available to the subject and to those around him. Specifically, reduction or elimination of stressors or altering the subject’s perception about the stress alleviates it and subsequent damage from it. For this reason as well as those mentioned above, the impact of using the stable model over traditional variants is a more accurate method in the situation provided and more effective at mitigating stress.

Conclusions

There are many models for stress, but no single one of them is perfect. However, the SSM comes quite close to achieving the ideal. While one could argue for the usage of traditional models under certain narrow circumstances, SSM offers a flexible approach to analyzing stress regardless of context. Learn this model, and the need to learn several models simply vanishes. Though the debate over theories of stress reduction remain, there is a model to encompass all of them and to build software that works against stress in emotional and physical situations.

Chapter 6: Influence Pattern as a Form of USER

Introduction

There are many situations in the modern world in which influence is a subtle but significant factor. Business and politics abound with a myriad of instances that must be analyzed and understood for the right course of action to be determined and followed. In this chapter, we look at a model for analyzing these situations without resorting to constructing entirely new software systems for each. The reuse of the core elements of the system will reduce the long term costs in both development and maintenance, and the resulting systems will be able to adapt to meet continuing needs.

Influence can be loosely defined as the act of someone or something affecting the process or outcome of an external entity. With so broad a definition, it should come as no surprise that influence is a significant component of practically any system, software or not. While the subtleties of influence are often hidden out of sight of the user, they are not entirely absent. In many systems, the impacts are so minor as to render the need for an accurate influence model excessive. For those instances in which influence plays a major role, it is best to have an accurate, flexible, and reusable model to improve the reliability and maintainability of the system.

There are certain issues involved with the current system of influence analysis, and these are explained through the use of a number of sample scenarios or contexts. There is a need for including influence into system design. Issues that the more traditional models have with flexibly for incorporating influence into their systems will be addressed.

Herein is introduced an alternative model as described by SSM, showing how such a solution can be applied to the scenarios described. This is followed by a comparison of the two systems, traditional and SSM, using qualitative and quantitative metrics with an eye for maintainability and reusability. Finally this chapter will conclude with a broader range of scenarios, demonstrating the many different types of situations that can benefit from the use of this influence model.

The Problem

One of the biggest and fastest emerging fields that deals with the nature of influence is data analytics. Companies spend vast amounts of money to analyze business data in order to better meet customer demands and thereby improve net revenue (Savitz, 2012). In this world of big data, each choice by each user has side effects that ripple throughout the entire system, influencing all future transactions. Influence can also be extrapolated to the business level in the physical world. How does an organization know that its advertising campaign is having the desired influence on the population? Such situations must be modeled to analyze influence.

The scenario types mentioned above are broad and differ based on company, system, available data, and a host of other indeterminate variables. Initially, it seems that each instance would require a custom solution, built almost entirely from scratch. Until now, this would probably be the case. A different model is needed to bridge the gap between these many disparate scenarios.

Discussion

As a demonstration, three scenarios are offered here. For the first scenario, let us consider a simple web-based advertising campaign. Consider a store chain, such as Target, that wants to increase its sales of sporting equipment over the summer months. The manager may choose to budget money to run an online advertising campaign.

Naturally, these expenditures will need to be budgeted and tracked for corporate headquarters. Moreover, it is incumbent on the manager to provide evidence of improved sales that were a result of the advertising campaign as proof that the funds were not misspent. The system should also allow for the use of third party companies that will create effective advertisements and publish them on sites and send them to individuals who are likely to be influenced to purchase Target's products. A sample model might look something like Figure 14.

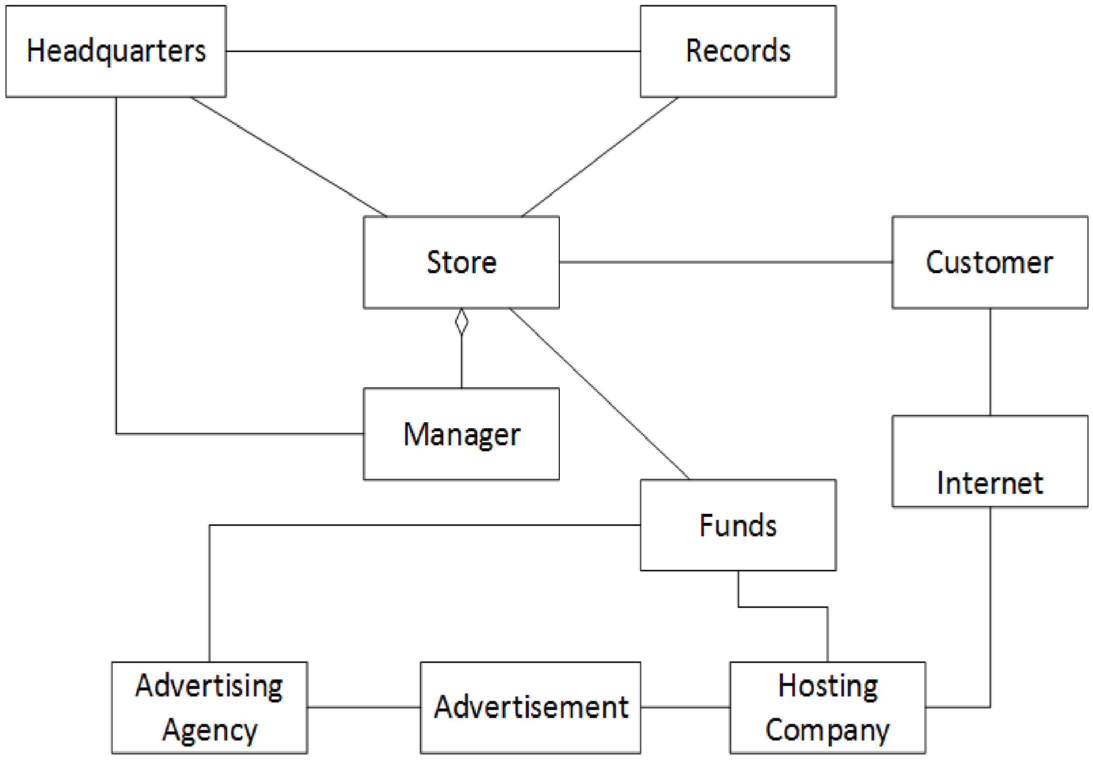

Figure 14. A generic model for web advertising.

In a different setting, one could also consider an FBI team tracking down a drug smuggling ring. The team uses informants to locate known dealers and then constructs profiles of those dealers to search for common denominators, or common spheres of influence. Locating one individual inexplicably linked to several dealers strongly implies a supplier whose traffic will be back-traced to locate processors and smugglers. Likewise, the operation may be done in reverse, but to get the whole ring, it is essential to first build the social network required to ensure the FBI locates all members. Such information can aid in inserting an undercover agent in the ring to gather incriminating evidence to ensure convictions. A sample model of how this might be represented and analyzed in software is shown in Figure 15.

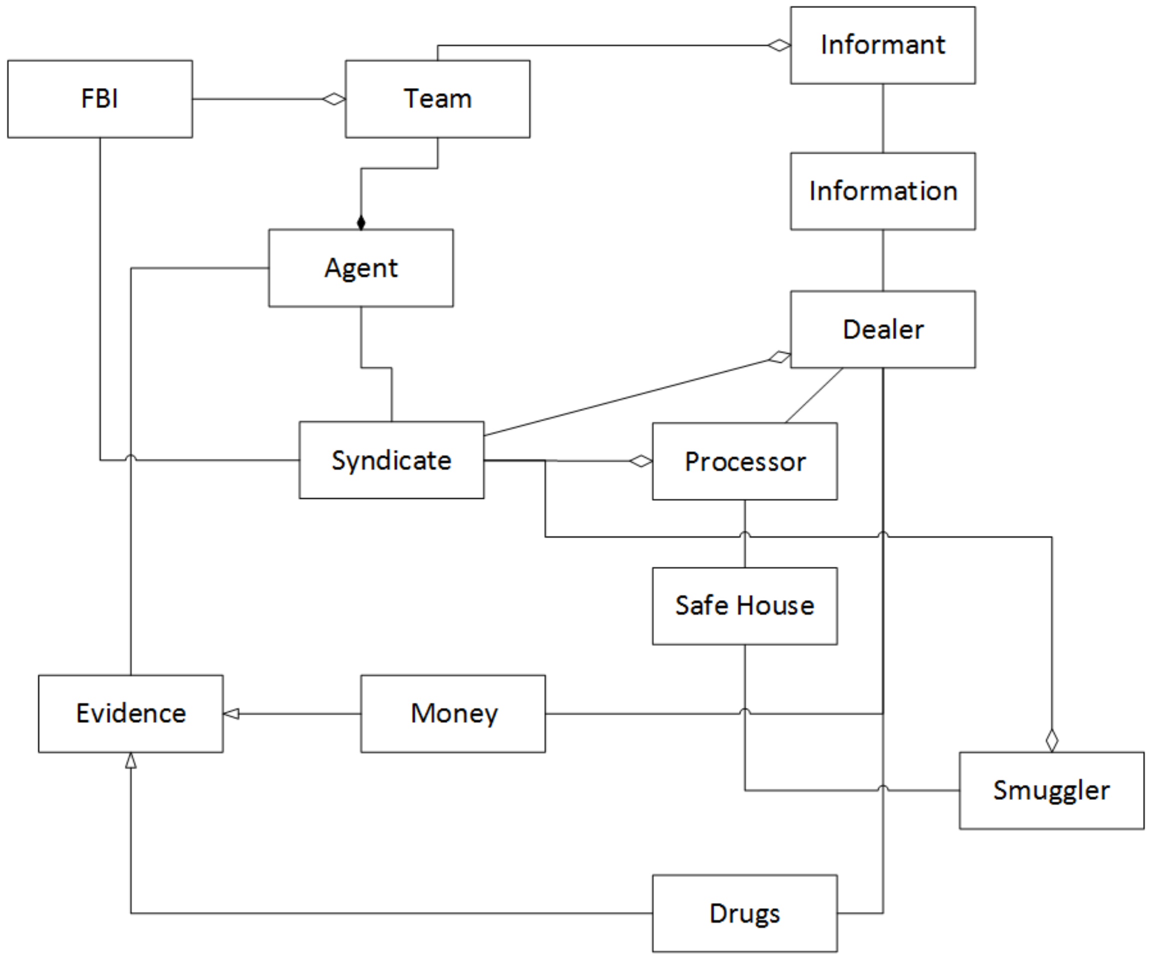

Figure 15. A model of an FBI sting operation.

Finally, consider an application of global proportions. In this hypothetical scenario, consider the ramifications of a regime change in an oil-producing country in the Middle

East, which for the sake of this illustration shall be called Stanistan. An immediate assessment must be completed, not only to calculate the shift in the balance of power for military and defense purposes, but also to determine if OPEC exports will be affected. The assessment must include the new leaders of the nation, their relationships to neighboring countries, the status of their economy, and the security they feel in maintaining their hold on power. Further consideration of probable actions by neighboring states will also factor into future economic or military plans. Naturally, with so many factors to consider, this is not a simple problem to solve no matter how one slices it, so for the moment only a solution for analyzing the situation, a model of which can be seen in Figure 16, will be presented.

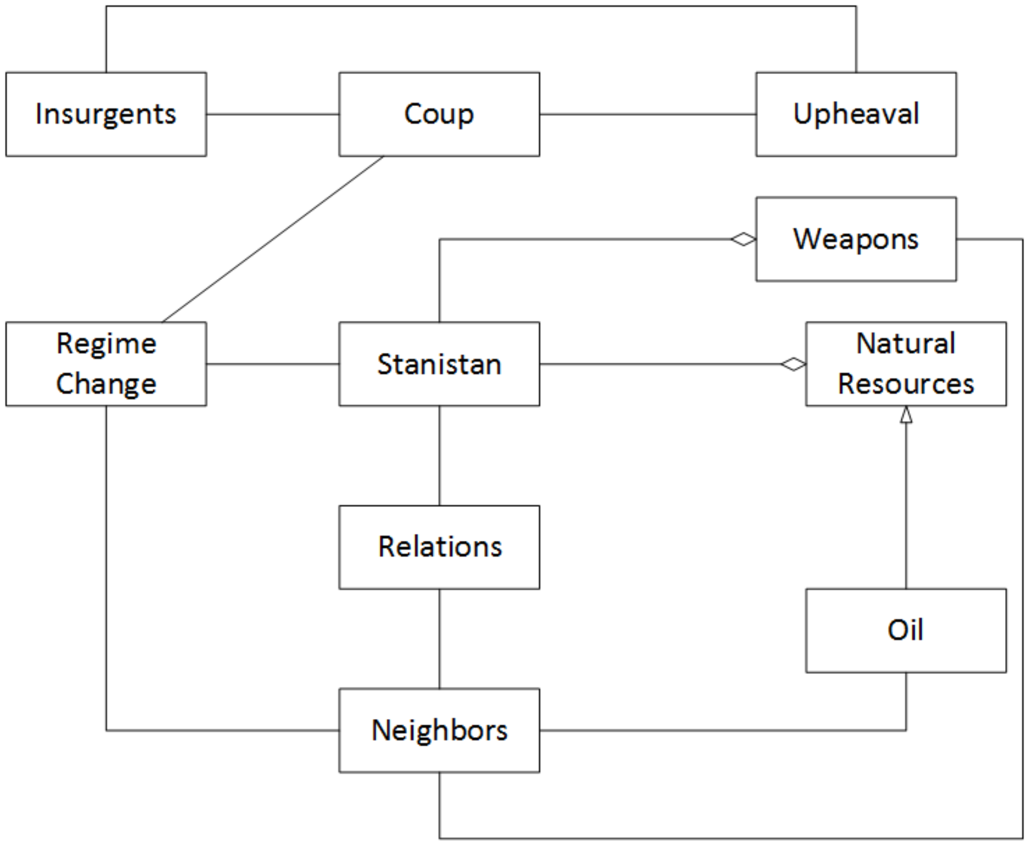

Figure 16. A model of a military coup.

Solution

In SSM, there are three tiers of objects to consider. The core concept of the model or pattern is an EBT, a concept that transcends the situation and simply is. For Influence, the ultimate objective is effectiveness because influence is the ability to cause an effect. There are a number of BOs that are more temporary in nature but are always an element of the model, which can be seen in Figure 17.

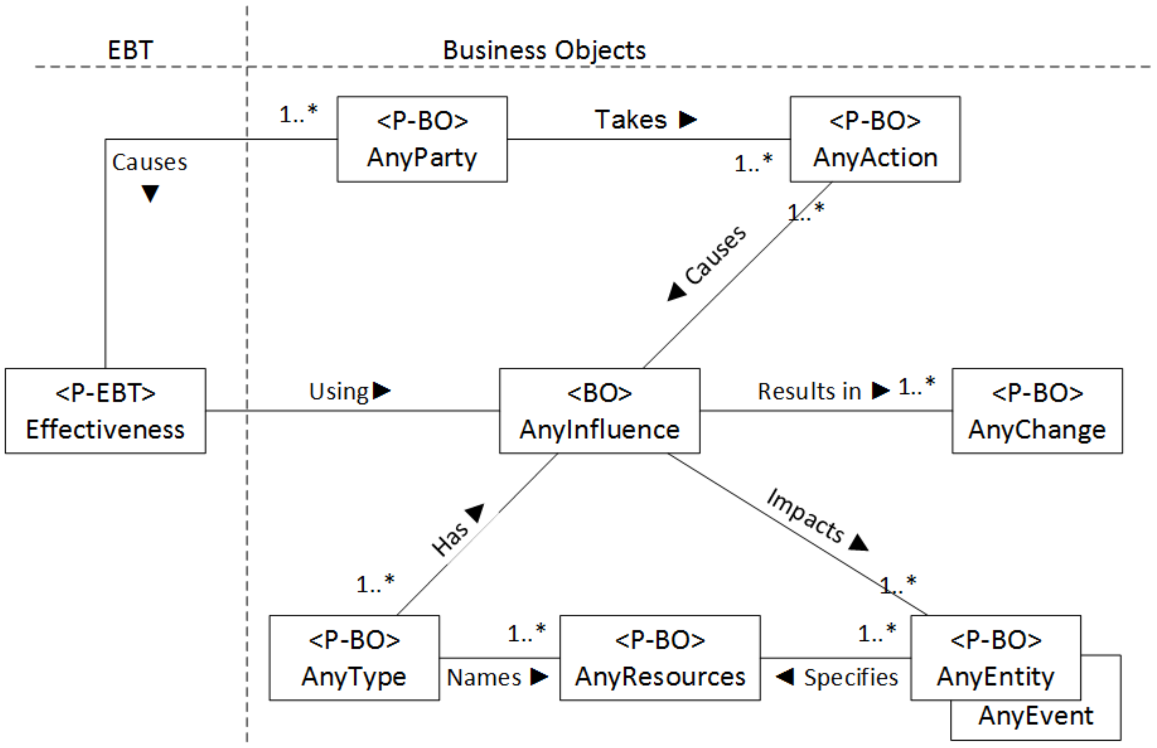

Figure 17. The stable software model for influence.

For example, in any situation there is always an action undertaken by a party (a legal entity). There is always a change as well as an impact on external events or entities. Each of these things is a general case that can be extended into the third tier of IOs that are added or removed based on the scenario. For example, the party could be Amazon.com and the action could be a special sale on a category of items. This scenario would operate on the same model as an entirely different scenario in which the NSA is the party and it is performing the action of tracking phone metadata.

In the Target sporting equipment sale in Figure 18, Target, the party, purchases influence in the form of an advertisement using an Advertising Agency, a Hosting Company, and the Internet. This impacts the customers to spend their money for sporting goods, producing increased sales and records thereof. This in turn makes shareholders happy and will likely liberate more funding for future use.

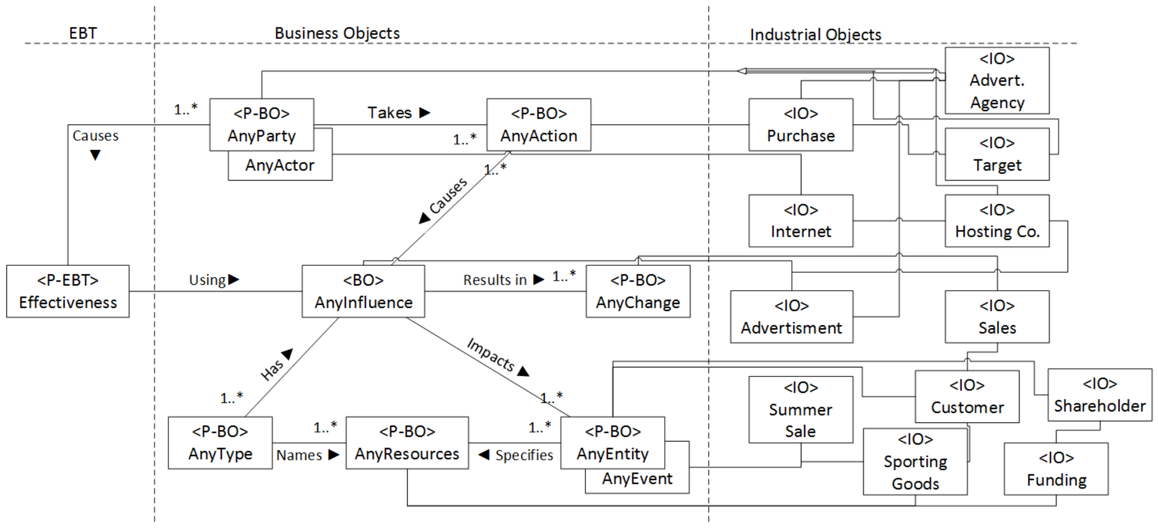

Figure 18. The stable model for Target advertising on the Internet.

In the FBI drug case shown in Figure 19, an FBI team, composed of agents, the parties of this scenario, extract information from informants, and use that information to successfully infiltrate of the criminal syndicate. The expected result of this is evidence in the form of money and drugs which will allow the FBI to arrest and charge the dealers, processors, and smugglers, and bring about quick convictions at trial. As an added bonus, all assets used in successfully prosecuted crimes, such as vehicles, safe houses, and other personal property are confiscated by the FBI and used to fund ongoing operations.

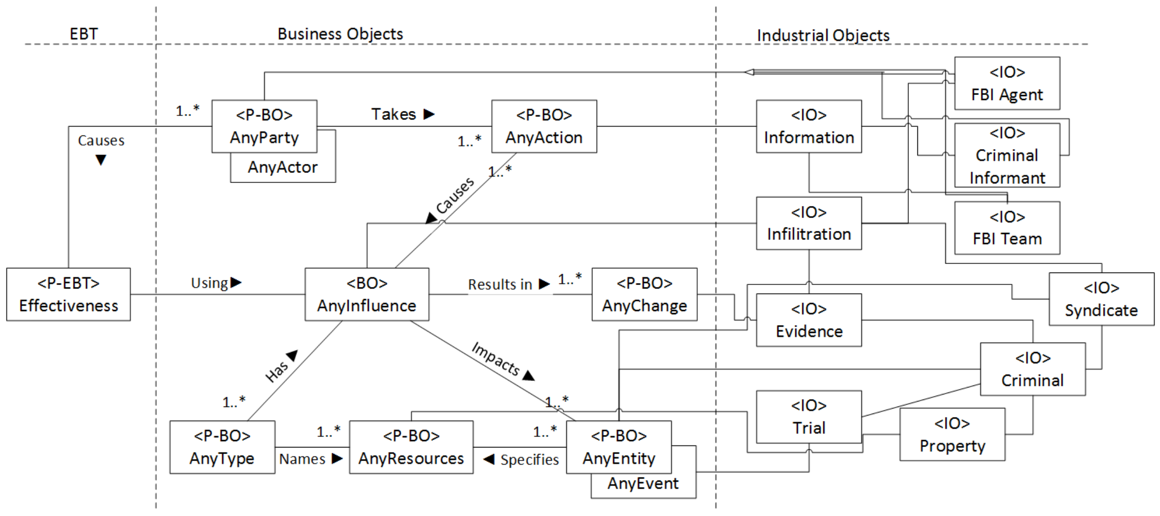

Figure 19. The stable model for an FBI sting operation.

In the unstable country scenario in Figure 20, insurgents execute a coup to oust the current leadership of the fictional country of Stanistan and cause a regime change. This will cause a general upheaval in the country which the insurgents will foster to ensure complete transition of power. This will also alter relationships with foreign powers, who may have supported or condemned the insurgents. The exact nature of the relationship with neighboring countries will depend upon their desire for the natural resources, such as oil, controlled by the new government and the weapons that government wields.

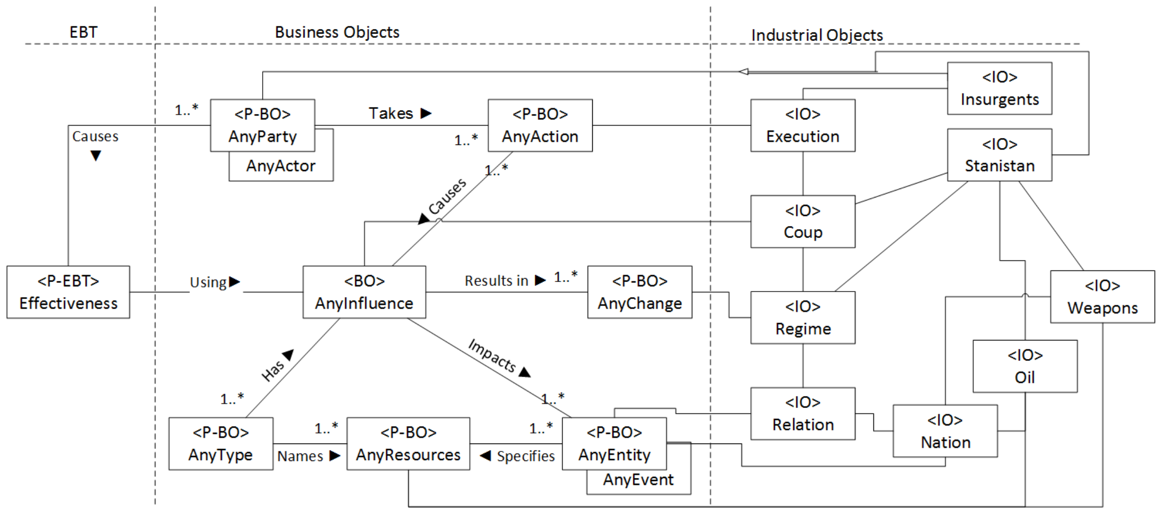

Figure 20. The stable model for a regime change.

Related Pattern

To measure and judge the effectiveness of SSM, one must have something to compare against. A sample solution would suffice but does not reach to level of a meta-model, which must describe the entirety of the problem rather than look at it through a single lens. Toward that end, an alternative meta-model is presented in Figure 21. This model is more expansive than the individual scenarios above, but still lacks utility compared to SSM.

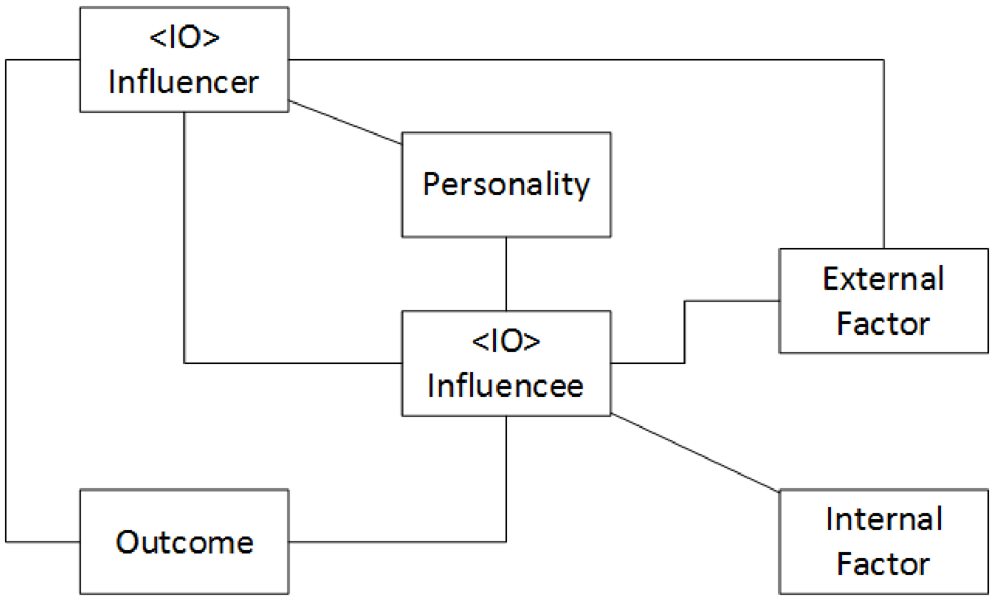

Figure 21. A generic model for influence.

Measurability

The two models can be measured in a number of ways, both qualitatively and quantitatively. A basic analysis of these two models is shown in Table 12.

Table 12.

Traditional/USER quantitative comparison for influence.

Feature	Traditional	SSM
Number of Tangible Classes	6	0
Number of Aggregations	0	0
Number of Attributes per class	5-7	3
Number of Operations per class	1-57	2-3
Number of Applications	1	Unlimited

Quantitative Measurability. The total number of methods in any system can be estimated using the formula: $T = C * M$. Where C is the number of classes and M the number of methods per class. A side-by-side comparison of these is presented in Table 13.

Table 13

Tradition/USER method count for influence.

Model	C	M	T
Traditional Influence Model	6	4	24
SSM Influence Design Pattern	10	2	20

In the initial analysis, the traditional model is already more complicated than the SSM variant, a situation that will only become more pronounced as subclasses are added to meet specific needs. Also, the use of tangible classes makes the traditional model subject to eventual obsolescence.

Qualitative Measurability. An additional metric could be the reusability of the classes involved. Obviously, the more reusable classes are in the system, the simpler and less costly the development and maintenance across multiple situations will be. Reusability can be quantified with the equation: $R = T_C – T_N$. Where T_C is the total number of classes and T_N is the number of classes not reused. Table 14 shows these results.

Table 14

Tradition/USER reusability comparison for influence.

Model	T_C	T_N	R
Traditional Model	10	8	1
SSM Design Pattern	9	9	1

The traditional metal-model is actually unable to reuse any of the classes since they must be replaced and extended to meet new scenarios. This differs from the SSM variant presented which reuses all base classes.

Applicability

As seen above, the SSM model for influence can apply to a wide range of different contexts. A more condensed, view of these disparate scenarios and others can be found in the Table 15.

Table 15

Applicability for influence stable design pattern.

Pattern objects	Web advertisement	Lobbyist	FBI drug sting	Regime change	Religious author
AnyParty	Target, advertiser, web host	Lobbyist, government official	FBI team, drug syndicate	Insurgents, Stanistan	C. S. Lewis, publisher
AnyActor	Internet	N/A	N/A	N/A	N/A
AnyAction	Buy advertising	Contribute to campaign	Gather information	Execute	Write books/stories
AnyInfluence	Web advertisement	Contribution	Infiltrate	Coup	Book/story
AnyResources	Funds, sporting goods	Power	Funds, vehicles	Oil, weapons	Words, ideas
AnyType	Financial	Political	Justice	Power	Allegorical
AnyEntity	Customers, share-holders	Vote	Dealers, smugglers	Foreign relationships	Minds
AnyEvent	Fourth of July sale	Vote on important bill	Trial	N/A	Book publication
AnyChange	Increased sales	Intended vote	Evidence acquisition	Regime	Reader’s opinion

Conclusion

As can be readily seen in the examples and associated metrics above, the most complicated and messy of situations may be analyzed by means of the SSM variant model for Influence. The SSM model transitions more effectively to entirely new scenarios and can, though not shown here, be used to locate weak points in a plan or possible solutions to a problem once the analysis is complete. The flexibility of the model is demonstrated both anecdotally and mathematically, making it a most reasonable to endorse it for use in any future systems relying heavily on the concept of influence.

Chapter 7: Using Reputation Stable Analysis Patterns as Model Based Software Reuse

"A good name is rather to be chosen than great riches…"
– Proverbs 22:1 (KJV)

Introduction

Reputation is the opinion (more technically, a social evaluation) of the public about a person, a group of people, or an organization. In other words, reputation is the general estimation that the public has for a person or an institution. It is an important factor in many fields, like business and online communities. It is also a subject of study in social, management and technological sciences. Its influence may range from competitive settings like markets to cooperative ones like firms, organizations, institutions and communities. Furthermore, reputation also acts on different levels of agency, individual and supra-individual. At the supra-individual level, it concerns groups, communities, collectives, and abstract social entities (such as firms, corporations, organizations, countries, cultures and civilizations). It affects phenomena of different scale from everyday life to relationships between nations. Reputation is a fundamental instrument of social order based upon distributed, spontaneous social control.

As can be seen, there is nothing different in the way reputation is handled in any of those areas of application. In fact, reputation is the same in all of them. Therefore, the reason for analyzing this concept with the sole purpose of extracting its core knowledge is worthwhile. This is even more important if one is planning to reuse it in numerous applications while still maintaining cost effectiveness.

The idea of reputation is commonly used in social life and economy, and there exists a common opinion on its general meaning. When it comes to a person, reputation is described as "a characteristic or attribute ascribed to one person (organization, community, etc.) by another person (or community)." (Dellarocas, C, 2003) On the other hand, the reputation of a service provider can be formed by means of a collection of ratings by different users. Each rating is intuitively equivalent to user satisfaction. The higher the rating from a user, the higher will be the reputation of the service provider.

Reputation is considered to be very relevant to systems in which there is information asymmetry about quality and trust due to the large number of players involved. Reputation can also be seen as a state variable that gives evidence about the missing information. Thus, reputation offers numerous incentives to providers and consumers to behave properly. Reputation provides a suitable mechanism to consumers to identify quality service providers and sellers. A reputation mechanism is quite successful when a steady-state market situation can be achieved and maintained.

The last decade witnessed an explosive growth in Internet connections around the globe. Online communities are gaining more popularity, as they neither limit nor restrict human interactions by insisting on geographic constraints. Instead, they bring together people of varied backgrounds, ethnicities, and nationalities. EBay, the largest person-to-person auction site, is an excellent example of such a community. Selling a product through such a community or becoming successful entrepreneur depends largely on the reputation of a person or an organization.

Reputation is a must in all types of businesses including online and e-commerce ventures. For example, Apple acquired a considerably good reputation by selling a high quality music player called an iPod, which eventually helped them to gain entry to the global cell phone market when they introduced the iPhone.

Traditional approaches to software design and development may not yield a stable and reusable model for gaining reputation. However, by using the SSM, software can be represented in any context by using a single model (Ahmed & Fayad, 2002; Fayad, 2017; Fayad, 2002a; Fayad, 2002b; Fayad & Altman, 2001). The SSM requires creation of a knowledge map by identifying underlying EBTs (Fayad, 2002; Fayad, 2017) and BOs. By adding IOs that are specific to each application and linking them with the appropriate BOs, the model can be applied to any application domain.

The resulting reputation pattern is quite stable, reusable, extendable, and highly adaptable. Thus, any number of applications can be built by using this common model. The reputation stable analysis pattern attempts to capture the core knowledge of reputation that is common to all application scenarios to emerge with a stable pattern (Fayad, 2017). The overall objective is to conceive and design a stability model for reputation by creating the knowledge map of reputation. This knowledge map or core knowledge can then serve as building block for modeling different applications in diverse domains (Savitz, 2012).

Pattern

The reputation analysis pattern abstracts this concept that can be applied to any party based on the any mechanisms. The reputation can be of any type and kind. This pattern also depicts the effect of reputation on the user. It is based on the principles of an EBT of reputation and is stable (Fayad, 2002; Fayad, 2017). It can be used to model any related scenario and is not restricted to any one scenario or situation. Hence, it is a stable analysis pattern.

Context

Individuals or organizations try to build a good reputation because of numerous corporate and business needs. A good reputation differentiates them from other organizations and drives more business to them. Further, an individual sees immense pride in attaining a good reputation. A bad reputation may also be fatal. For example, the bad reputation attained by ENRON meant them losing their business and led to filing bankruptcy. Hence, reputation is a crucial factor for any business.

Reputation is important enough to consider modeling in a number of applications concerning various organizations and institutions. Web portals like eBay and Google, personalities like Tiger Woods and George W. Bush, companies like Apple computers and CISCO, and countries like Switzerland and Saudi Arabia are all widely and well known, whether for good or otherwise.

Scenario #1: Reputation in online business. In an online business, such as eBay, reputation is a vital component of doing business. In this scenario, a seller and buyer use eBay, the mechanism in this scenario, to sell a product. The sale is usually contingent on the satisfaction of others shown as a positive rating from the opinions of others.

Scenario #2: Reputation in politics. Consider the political reputation of President G. W. Bush over the course of his presidency. In this situation, the media was used to impugn the trustworthiness of the President over the Middle East Conflict. This led to vilification and low approval ratings from the American public.

Scenario #3: Reputation in product marketing. In a marketing scenario, few can match the success of Apple with its most popular media player, the iPod. Apple is a market leader with significant revenue due to the demand of its customers in this business. This demand is in large part due to Apple's reputation for quality products.

Scenario #4: Reputation in corporate corruption. In the case of Solyndra, reputation had both political and business facets. In this situation, Solyndra received a grant from the government. However, the fiscal malfeasance in distributing the funds lead to significant public attention leaving the company in a vilified state and its executives untrusted when Bankruptcy occurred.

Scenario #5: Reputation in sports. The rise and fall of Tiger Woods demonstrates personal reputation and its importance. Through his inappropriate actions, vis-à-vis extramarital affairs, he proved his disloyalty to his family and fans. Through a number of press releases and events, Tiger Woods admitted to his transgressions, sparking a state of outrage and negative approval.

Problem

Today, global competition has already increased enormously. From simple pencils to gigantic airplanes, there are numerous players competing with each other to sell their products. In such a scenario, it is very much required that a person or organization develop a reputation in order to compete effectively with others. This can be done in a variety of different ways from selling quality, low cost products to developing the skills that others want to utilize. In an online business, there are numerous ways to develop a reputation such as through online feedback and rating mechanisms. Building a generic pattern that covers all such cases of reputation development is a challenging task.

Reputation can be applied to different domains, such as politics and business, and to different parties, such as an individual, an organization, or a country. Hence, it is very much essential to model a generic pattern. Criteria must be satisfied before developing a generic reputation pattern. The pattern should be reusable to model any reputation application or scenario. A thorough understanding of the core concept of reputation is absolutely essential so that the core knowledge can be properly captured.

Since reputation is used in different contexts and in different domains, building a generic model without loss of functionality is uniquely challenging. By using SSM, this problem is solved and a generic model is created accommodating the various domains. This model is illustrated and described in the solution section.

Solution

The solution shown here utilizes SSM to explain the concept of reputation (Ahmed & Fayad, 2002; Fayad, 2017; Fayad, 2002a; Fayad, 2002b; Fayad & Altman, 2001). Figure 22 depicts the class diagram for Personalization pattern. There are a number of participants of the Reputation pattern.

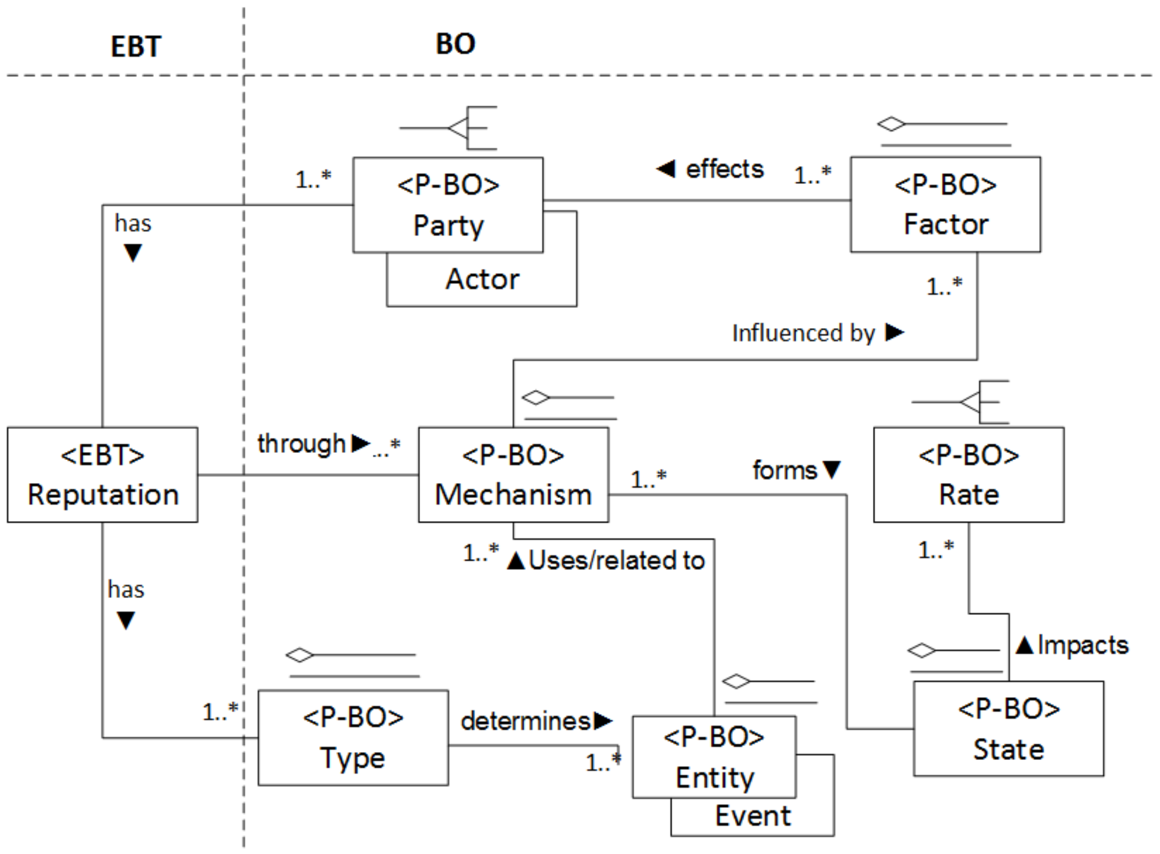

Figure 22. Reputation stable analysis pattern

Classes:

- Reputation represents the reputation. It is an EBT that presents the enduring and business knowledge, which discloses relevant information based on the attributes of the user.

Patterns**:**

- AnyParty represents any person, individual, an organization, or group with whom the reputation is associated.
- AnyFactor denotes the factors that affect the reputation of a particular individual or organization.

- AnyEntity denotes the characteristic or the product for which an individual or organization is reputed.
- AnyMechanism denotes the methodology through which the individual or the organization achieves the reputation.
- AnyState denotes the position achieved by any individual or an organization when applying the mechanism
- AnyRate represents the status that was achieved by applying the mechanism and that was impacted by the state.
- AnyType represents the nature of reputation achieved by any individual or organization.

The class diagram in Figure 22 provides visual illustration of all the classes in the model along with their relationships with other classes:

- Reputation is the EBT of this pattern and is associated to AnyParty (BO).
- Reputation (EBT), which has AnyType (BO), is achieved through AnyMechanism (BO).
- AnyMechanism (BO) uses AnyEnity (BO) to achieve Reputation (EBT).
- AnyParty (BO) chooses AnyMechanism (BO) because of the influence created by AnyFactor (BO).
- AnyMechanism (BO) forms AnyState (BO) and leads to AnyRate (BO).
- AnyFactor (BO) affects AnyParty (BO) leading to Reputation (EBT).

Consequence

Using the reputation analysis pattern will require or demand that the entity has correct attributes available on which to base the reputation. Also, the preferences of AnyParty involved in the reputation process must be immediately made available. However, this does not mean that the pattern is incomplete as this is the nature of patterns. They need to be used with other components.

The good thing with the reputation analysis pattern is that the pattern has been derived with stability in mind. It has captured the enduring knowledge of business and its capabilities and will stand the test of time. However, the bad thing about it is that it might result in incorrect or inaccurate results when reputation is not performed in a proper manner. In addition, the privacy of AnyParty might be invaded when trying to collect attributes for AnyParty. The reputation pattern has a number of benefits:

- Flexibility—this reputation pattern is very flexible and highly adaptable, as per the preferences of AnyParty (Fayad & Cline, 1996). As the preferences change the reputation of the final product can be easily altered.
- Reusability—the reputation pattern is a very stable pattern. It can be reused in many different scenarios spread across many different fields (Mahdy, Fayad, Hamza, & Tugnawat, 2002).

Applicability

Table 16 depicts a five potential scenarios in which reputation may apply to software. In this section, two of these examples are discussed to further illustrate the use of reputation analysis pattern.

Table 16

Potential applications for applicability.

Pattern objects	eBay	President	iPod	Solyndra	T. Woods
AnyParty	Seller, buyer	President, citizen	Apple, customers	Solyndra, government, the public	Woods, fans, family
AnyActor					
AnyFactor	Satisfaction	Trust	Demand	Money	Disloyalty
AnyMechanism	eBay	Media	iPod	Fiscal Malfeasance	Affair
AnyRate	Positive rating	Approval rating	Revenue	Attention	Negative approval
AnyState	Opinion	Vilification	Market leader	Vilification	Outrage
AnyEvent		Iraq war		Bankruptcy	Press release
AnyEntity	Product		iPod		Mistress
AnyType	Business	Political	Business	Business, political	Personal

Application #1: Reputation in Online Sales. In the context of eBay, selling a product online and obtaining a good recommendation from the buyers through the eBay feedback system is called reputation building. The eBay website allows users to enter their ratings on various categories. The model for this application is shown below in Figure 23.

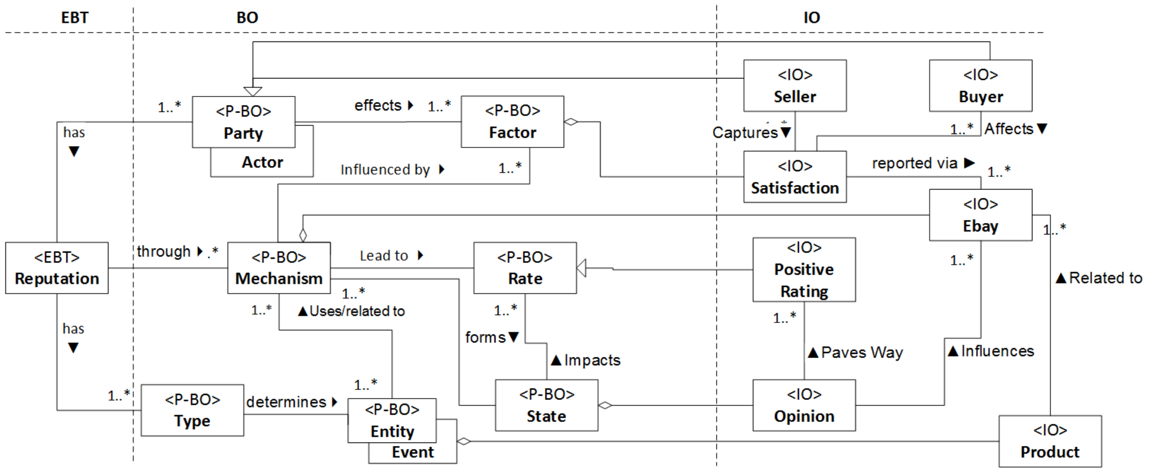

Figure 23. Reputation analysis pattern for online eBay sales.

Class Diagram Description:

- AnyParty (Seller) sells AnyEntity (Product) through AnyMechanism (eBay).
- AnyMechanism (eBay) is searched by AnyParty (Buyer), who finds the AnyEntity (Product) sold by AnyParty (Seller) suitable for him.
- AnyParty (Buyer) buys AnyEntity (Product).
- AnyParty (Buyer) likes the AnyEntity (Product), which creates AnyFactor (Satisfaction) in AnyParty (Buyer).
- AnyParty (Buyer) through AnyMechanism (Feedback) provided on the eBay website records his or her AnyState (Opinion).
- AnyState (Opinion) impacts AnyRate (Positive Rating) given to AnyParty (Seller).
- AnyRate (Positive Rating) provides a good Reputation to AnyParty (Seller).

Application #2: Reputation in Apple, Inc. Apple, Inc., is famous for its Mac PC’s. When the company’s shares were dropping, Apple, Inc., re-established its reputation by developing a quality music player called iPod and by selling them. The following application models how Apple re-established its reputation. The model for this application is shown below in Figure 24.

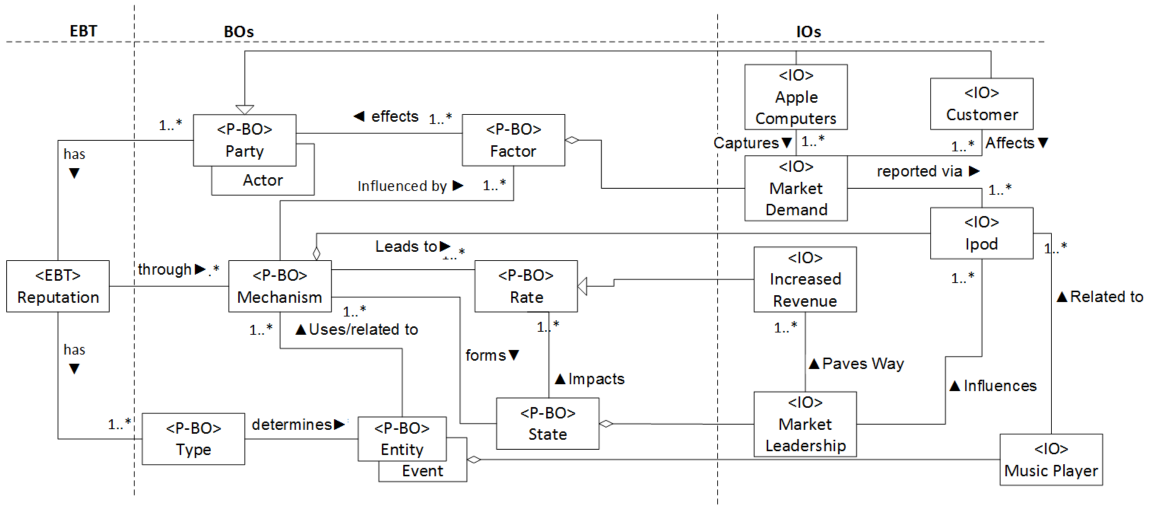

Figure 24. Reputation analysis pattern for sale of Apple iPods.

Class Diagram Description.

- AnyParty (Apple, Inc.) captured AnyFactor (Market Demand) for AnyEntity (Music Player).
- AnyFactor (Market Demand) is created by AnyParty (Customer).
- AnyParty (Customer) is looking for a good quality AnyEntity (Music Player).
- AnyEntity (Music Player) details are obtained by AnyParty (Apple, Inc.).
- AnyParty (Apple, Inc.) develops AnyMechanism (iPod).
- AnyMechanism (iPod) is bought by AnyParty (Customer).
- AnyParty (Customer) is influenced by AnyFactor (Quality) of AnyMechanism (iPod).
- AnyFactor (Quality) helps in AnyParty (Apple, Inc.) in gaining reputation by increased AnyRate (Revenue).

Related Patterns and Measurability

The reputation stability pattern is generalized enough to allow for its applicability in diverse application domains. This pattern includes EBTs and BOs, so its applicability in other related domains just requires the attachment of IOs on peripheral boundaries. This pattern is quite complex in design, and it requires deeper analysis to identify key EBTs and BOs. However, it greatly enhances pattern reuse and effectiveness of finding a practical solution.

The traditional model, as shown in Figure 25, is based on IOs. Recall that IOs are physical objects and are thus unstable. The traditional model caters to the current requirements. The traditional model is also hard to reuse when those requirements change. Any change in the requirements requires complete reengineering of the project. Hence, the traditional model involves a high maintenance cost in terms of time, labor, and money because the system built by using traditional model cannot be easily extended or adapted.

Traditional Class Diagram. The reputation traditional model is based on IOs, which are non-enduring and non-adaptable objects. Any change in a single IO may initiate a cascade of changes throughout other IOs, making it highly unstable. This model cannot remain stable for an extended time span, whereas the reputation stability pattern can because it is based on enduring concepts (e.g., EBTs and BOs), which are adaptable and durable. This confirms its continuous applicability.

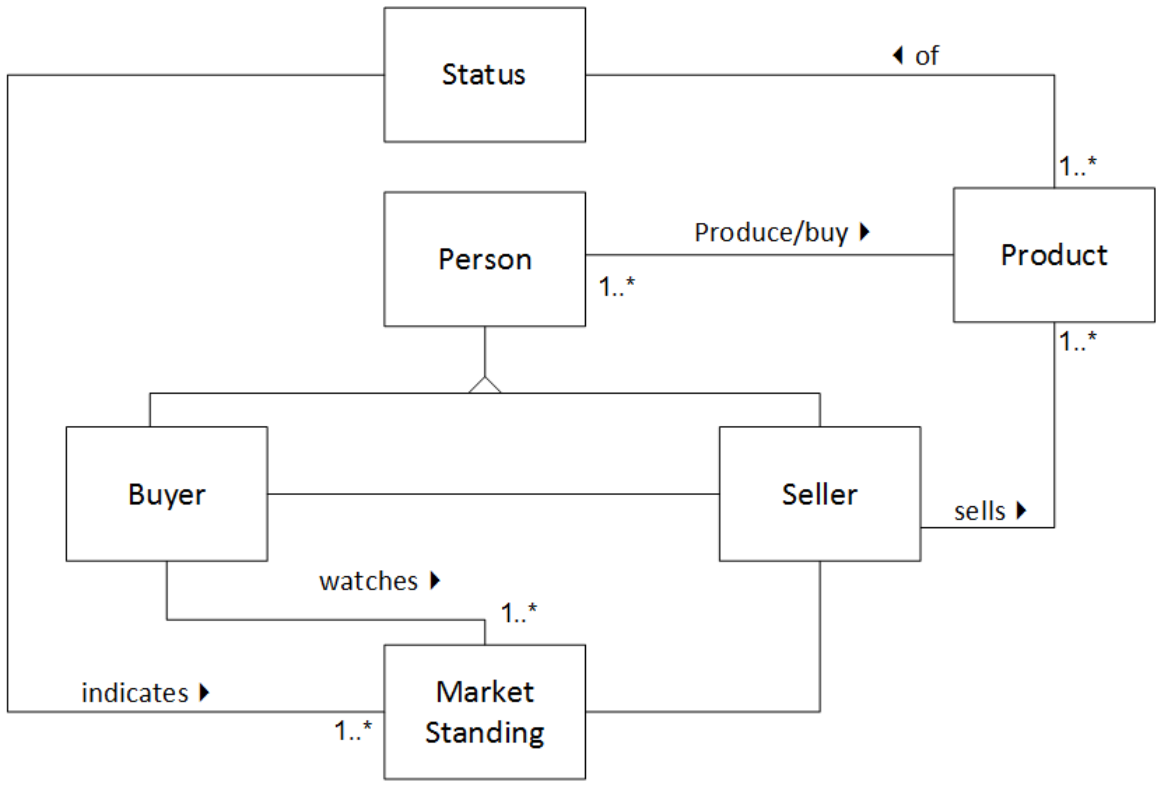

Figure 25. A traditional model for reputation.

Applicability of the traditional model is limited to a particular domain area. In the case of the reputation traditional model, it is tied to reputation of one product and one company. On the other side, a stability model built on the reputation principle is applicable to a number of domain areas that have many core themes in common. Hooks can be easily used to extend and reuse this stabilized model.

The identification of objects involved in the traditional model requires brief knowledge and documented data about the specific domain. These objects can be easily found in a problem statement. But in a stable model, one requires deeper study, experience with the domain, and intuition to come up with a useful set of EBT's and BO's.

Quantitative Measurability. A simple evaluation based on the number of classes is easily made. In the traditional model, there are a mere six classes to implement as opposed to the seventeen required to set up the alternative SSM. However, this also means that the code would be more spread out in the SSM variant, allowing for looser coupling between classes. Additionally, ten of the seventeen classes are easily used in other applications, meaning that a large fraction of the code may already be written. As an added benefit, the SSM provides a ready framework for the application so that it does not need to be coded from scratch.

The traditional model shown above is applicable only to a single domain and use. Arguments could be made that subtle adjustments may make the code adaptable to closely related scenarios, but the more distinct the scenario, the messier the results will be, resulting in substantially increased maintenance costs. With SSM, on the other hand, once the initial coding for the core concepts is complete, they can be reused in applications across many domains, making SSM the superior choice.

Qualitative Measurability. The greatest benefit that SSM has over traditional models is that it is adaptable to many circumstances (Mahdy et al., 2002). To demonstrate, one can compare the models for code reuse. The stable software model for this scenario has 17 classes of which 10 are reusable with little to no modification, while with the traditional model one would be lucky to reuse parts of one or two classes while converting to a moderately similar application. The traditional model's reusability is therefore 0%, while the SSM is always at least 59% reusable for any applications of reputation.

Summary

The Reputation pattern proposed here is based on the principles of stable analysis pattern. The pattern is explained with two applications that perform well based on this model. The depth of this reputation pattern depends on the availability of AnyParty's attributes for personalizing the particular application. Each object in the reputation pattern has its own role and play, which is independent of any applications, where this pattern will be applied. More than one mechanism exists to carry out the reputation. Care should be taken, while choosing the appropriate mechanism by utilizing the attributes properly.

One difficult part of modeling a reputation problem was finding a good class diagram description. Making the description as clear and accurate as possible so that it is beneficial in drawing the sequence diagram is the key to getting a good model. The process of creating the sequence then gets much simpler and flexible, as it is just the translation of the class diagram.

Though building a stable design pattern for reputation that can be reused and reapplied across diverse domains is difficult and requires a complete understanding of the problem, it is worth the effort, money, and time. Modeling the reputation pattern by using SSM results in a reusable, extensible, and stable pattern.

This pattern is so flexible that it can be applied to any type of scenario. The industrial objects can be hooked to the business objects to make it more meaningful to the scenario where it is applied. However, the correct identification of EBT and BOs for reputation is the most challenging task and requires some prior experience. Once the EBT and BOs are correctly identified, the next challenge is to determine the relationship between the EBT and BOs so that the reputation pattern can hold true in any context. Once this is ascertained, depending on the application, the IOs are attached to the hooks provided by BOs. Thus, by using the reputation pattern as a foundation, an infinite number of applications can be built, just by plugging the application specific IOs into the pattern. This results in reduced cost, reduced effort, and a stable solution. Hence, the reputation design pattern is very useful and beneficial to developers as well as users.

Chapter 8: USER using Stable Analysis, Design and Architectural Patterns

Introduction to SSM

The endless pursuit of creating effective systems for software reuse has continued for as long as software has existed. To date, there have been few effective systems created for ensuring a high degree of reusability from one project to the next. The inherent tendency for projects to demand substantial alterations despite being designed for maximum reusability remains strong evidence of this fact.

Software reusability makes the study of stable analysis, design, and architecture patterns a domain of immense interest. By extrapolating the stable concepts that use SSM and knowledge maps, one can realize software solutions that do not need excessive alterations, changes, or additions. Such patterns function as a framework, to which new objects can be added depending only on the uniqueness of the scenario to which it is applied.

Patterns are models that are reusable in the future. The problem with many existing software engineering patterns is that they are generally domain dependent. Using them for entirely different applications could be very hard because some modifications or changes will have to be made. Using software stability concepts to generate patterns promotes greater reuse as stable models, which use EBTs and BOs, are created for keeping the goal of a system in mind (Fayad, 2002a; Fayad, 2002b; Hamza, 2002). The goal of a system, which forms the foundation of the pattern rarely changes. Subsequently, the models generated from them are more stable. This stability means that the pattern itself never changes, although it is extendable for use in various applications regardless of the domain.

Stable patterns always depict concepts and theories (Fayad, 2017). Concepts are more likely to be implicit, and they are always associated with semi-tangible objects, through which they will become explicit. The concept could be inherent in the objects, it could represent a property of the object, or it could represent a relationship between the objects. Through the pattern, one can recognize the objects that always appear with the concept. However, the object abstractions that are created should be extendable in a manner that allows them to be application independent. Using the software stability concepts, it becomes very convenient to model patterns that do not change over time, as they encapsulate the core concept of a system. The concepts modeled as EBTs and the semi-tangible objects associated with the concepts modeled as business objects (BO's) are the foundations on which the stability aspects are developed. This is extended to industrial objects (IOs) through hooks or extension points, such as the patterns of Gang of Four, Simons Group, etc.

Figure 26 depicts the three layers of the SSM (Fayad, 2002; Fayad, 2005; Fayad & Altman, 2001) and the relationship of the EBT, BO, and IO Layers.

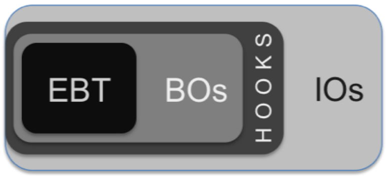

Figure 26. SSM Structure.

In SSM, no pattern is isolated from other design and analysis patterns. Each pattern is composed of EBT and BO objects which have their own associated patterns. Accordingly, in models and figures displaying analysis, design, or architecture patterns, the presence of a secondary pattern will be shown along with the type of object (e.g. <P-BO> means a business object that is also a pattern). These model objects will be appended with the word “Any”, in reference to the broad range of options available. For ease of reading, however, all business objects will merely be highlighted and dropped rather than prepended.

Stable Analysis Patterns

Stable analysis patterns (SAPs) are synthesized based on the concept of an EBT. These EBTs are intangible, implicit to a given scenario, and extremely stable over time. An EBT is described in the form of an SAP by using some common non-tangible concepts, known as business objects. An example of an EBT is the concept of reputation. This concept can be represented as a model then implemented in software to apply to many contexts in which reputation is a needed factor.

Sample Applications. In an online business scenario, such as eBay or Amazon, reputation is a critical component for carrying out profitable business operations. In this scenario, a seller and buyer use eBay or Amazon as their mechanism to sell a product. The sale is usually contingent, reflected in the factor of satisfaction of others buyers which they express as a positive rating through their opinions of others.

In the domain of personal music player marketing of the popular media player, iPod, Apple is the undisputed market leader and is known to generate significant annual revenue due to the very high demand, by customers, within the said business segment. This demand is based on Apple’s reputation for producing high-quality players.

Sample Solutions. Using the two examples above, it is possible to model workable solutions for the given contexts. These solutions can be found in Figures 24 and 25 in chapter 7, respectively.

Stable Design Patterns

The main distinction between a stable design pattern and a Stable analysis pattern is that the design pattern is focused on or built from a business object, rather than an EBT. Business objects differ from EBTs in their temporal nature, having a beginning and an end. Beyond this, similar rules for representation apply.

It is important to note that stable design patterns are quite different from the "design patterns" that are typically discussed and attributed to the Gang of Four. While such patterns are used as strategy, observer, and decorator to demonstrate patterns in problem-solving techniques, they are conceptually distinct from the stable design pattern, which instead, abstracts an entire set of situations or scenarios, rather than problem-solving practices. To illustrate stable design patterns, one can use as an example the concept of influence.

Sample Applications. Consider the case of a company choosing advertising on the web to influence the spending habits of the online shoppers. The business will begin by first negotiating a contract with an advertising firm, who will assist in this endeavor, for a negotiated price. The advertising firm will then assist the company in making short videos, banners, and website sidebar. Once the content is ready, the advertising firm places the advertisement in their database for future use by an analytics engine, which will determine websites where the advertisements are likely to perform better. The company that wants advertisement services will be billed according to the terms of the contract, which will primarily depend on consumer web response, measured in views, hits, or rollover time. These statistics will be combined with sales data to inform the companies how effective the advertising was, and it will eventually help them make decisions concerning any future advertisements.

The role of influence can also be clearly seen in global politics as well. Internal instability in a country rich with natural resources but little else can lead to some serious complications on the world stage should a coup occur. In such circumstances, rebel factions can overtake a government to gain power over its resources and people. This impacts all other governments, especially those for whom trade is desirable or even necessary. This then opens the possibility for further destabilization if other countries believe the new regime is too strongly against their best interests.

Sample Solutions

Using the two examples above, it is possible to model workable solutions for the given contexts. These solutions can be found in Figures 18 and 20 in chapter 6, respectively.

Stable Architecture Patterns

Stable Architecture Patterns is the end result of blending two or more design or analysis patterns. This is done simply by merging the core concepts shared between patterns, so that classes (such as Actor and Party) are not duplicated. For an example of a Stable Architecture Pattern, consider Conflict Analysis, in which analysis is the EBT and conflict, being more temporal, is a BO.

Sample Applications. Conflict analysis can have many forms depending on its source of origin. Though there is infinite variety in the types of conflicts that are available for analysis, some of the more accessible ones may arise from highly publicized reality TV shows, where legitimate personal issues are raised to provide entertainment to the viewing public. For example, midway through the filming season five of Teen Mom, the MTV producers suddenly changed their opinions of removing Farrah Abraham from the show because of her other commitments and later invited her back perform again to act in the series. This act made another cast member upset, and the actor decided to terminate her portion of acting because of the confusion caused by differing viewpoints on some of the external issues. To dispel bitter acrimony, the situation was reviewed again by the cast member, who eventually decided to remove herself from filming process, but agreed with the MTV producer to continue the show without her son.

A conflict arises due to natural complexities involved in the process, and most of them are related to interpersonal and emotional issues of the actors involved. Another classical scene would be a complicated labor strike in an auto spare parts supplier unit for the GM; wage reductions could be one of the main reasons for the strike. In this example, an attempt by the spare parts manufacturing company American Axle to slash labor costs (including pension and health care benefits) by as much as 50% caused an unprecedented uproar in the United Auto Workers Union. As a result, auto supplies to almost thirty facilities that manufactured GMC vehicles were seriously affected. Eventually, these facilities started operating on just a single shift, or they were closed indefinitely.

Sample Solutions. In the example of the MTV interpersonal conflict, the story begins with MTV and its Cast Members, which are all legal parties in this scenario. When considering the aspect of their careers, an analysis of the conditions of employment defined by contracts should result in a plan of action, or consequence. This plan will need to be developed at meetings with both the producer and editor and must account for factors such as revenue and might ultimately affect subsequent filming sessions of later episodes and future seasons. A fully modeled solution is given in Figure 27.

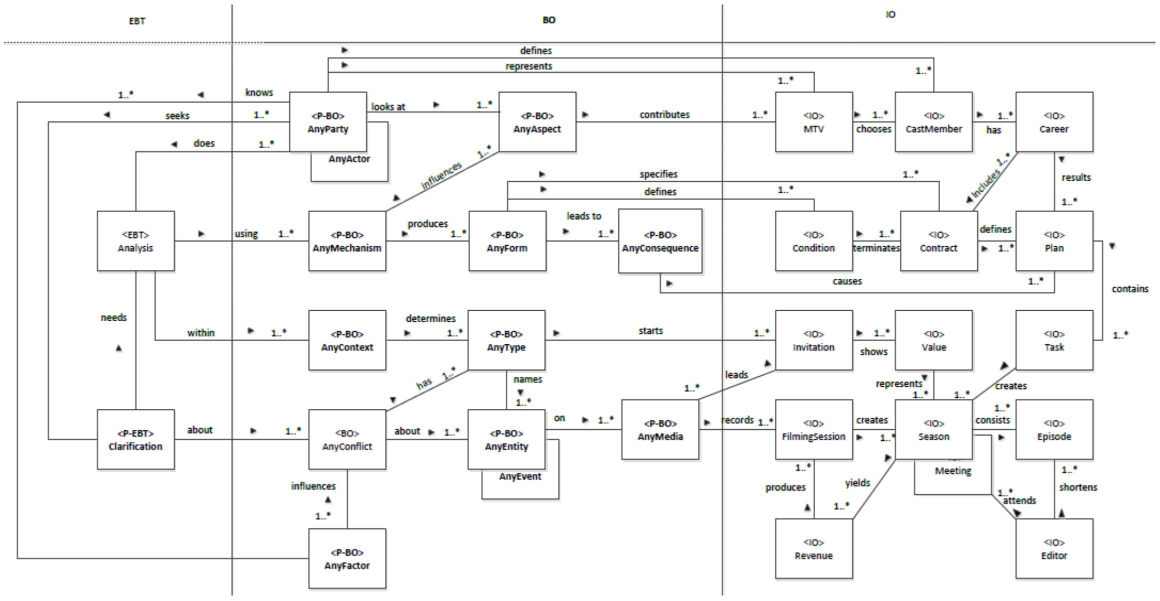

Figure 27. Conflict analysis of reality TV show.

In the case of the auto parts manufacturer strike, the primary parties involved include workers and other employees, the company board members and the labor union. The workers, due to the factor of inflation, submit their demands in some form to management and decide to strike and boycott working, unless they are given a raise. The labor union, in accordance with the Labor Beneficiary Act, enters into negotiation for a new contract. In this case, the strike is the conflict, the raise is the intended consequence, the Labor Beneficiary Act provides the context for the negotiations which are the event, and the contract is the media. This solution is modeled in Figure 28 below.

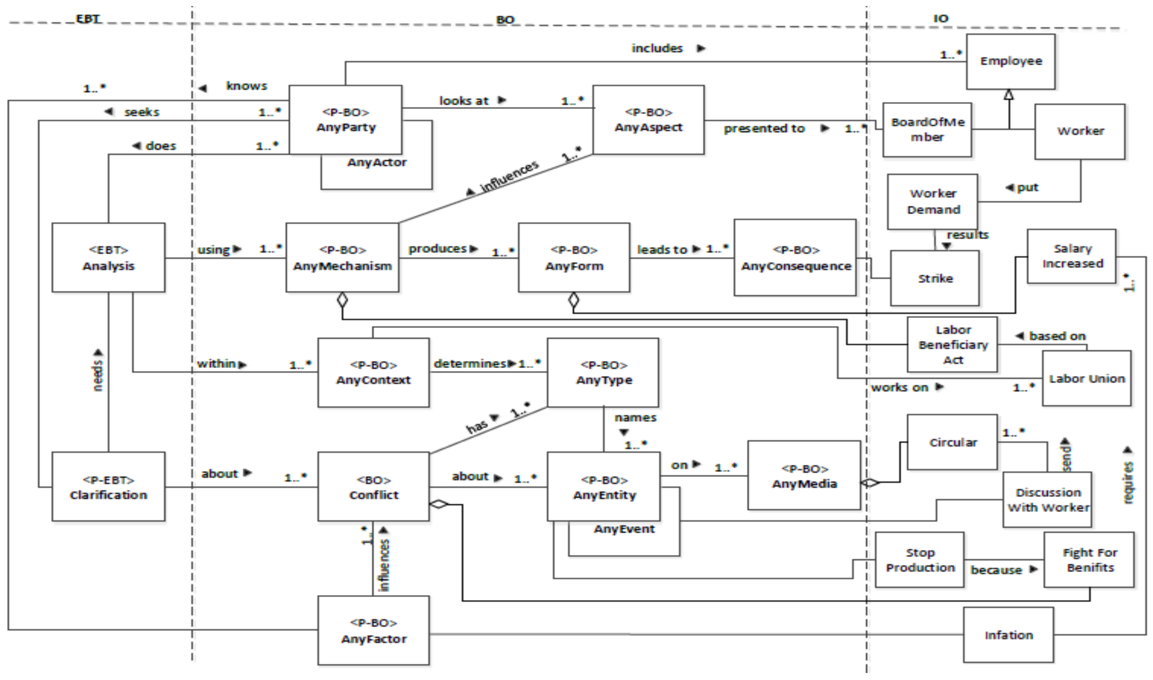

Figure 28. Conflict analysis of a labor union strike.

These models can be compared with the traditional model of conflict analysis which is shown below in Figure 29. This makes an effective general case for scenarios involving government mediation, eminent domain seizure, for example. But in other scenarios, this model is decidedly less useful. This is the natural consequence of using a traditional model, which cannot be easily reused like a Stable Software Model.

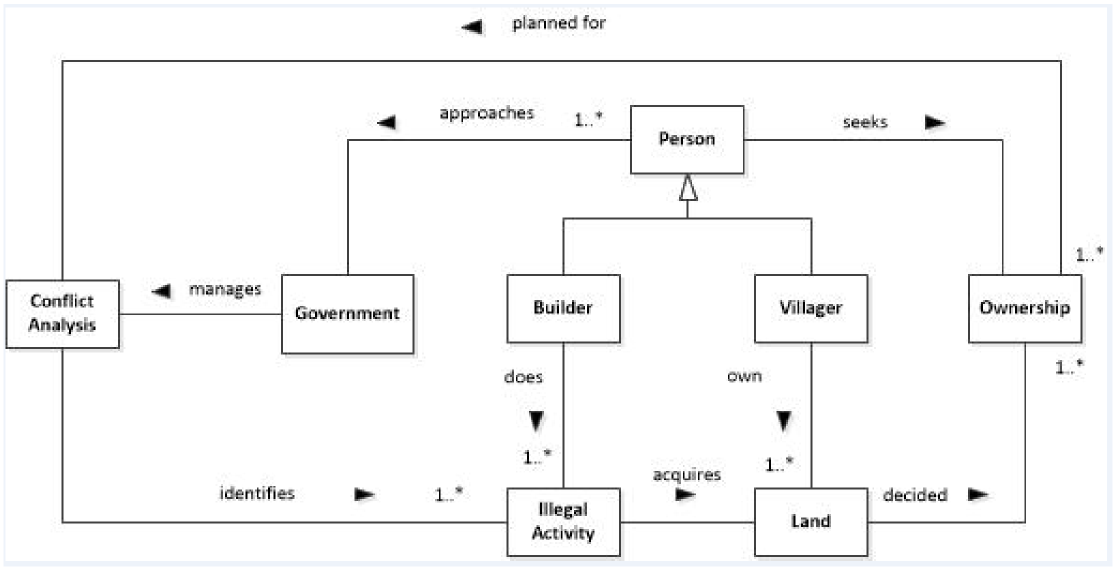

Figure 29. A traditional model for conflict analysis.

Analysis. In the simple examples given above, it should be noted that SSM, though requiring more initial work than the traditional model, is also more widely applicable to new and different contexts. For more refined evidence, a study of few relevant metrics might be helpful.

Quantitative Measurements. Perhaps the easiest measurement is reusable class count. In the SDP example of web advertisement, there are eight classes in the traditional model, of which none are applicable to the alternative influence example of a family debate over sports. The SSM model, in contrast, shares all nine core classes (EBTs and BOs) across both scenarios. Likewise in the SAP examples involving product and vendor reputation, the traditional model does not have any adaptable classes that could make it stable and extendable. The SSM model, on the other hand, can easily maintain their core classes with newer scenarios.

There are additional potential metrics, cyclomatic complexity, for example. This metric is used to determine the interconnectivity of the objects in a given design or model. The calculations are included in Table 17, given the equation: $M = E - N + 2P$. Where M is the cyclomatic complexity, E is the number of edges (connections between classes), N is the number of classes, and P is the number of connected components, which for software models is almost always one.

Table 17

Cyclomatic complexity of influence models.

Model	E	N	P	M
Traditional Influence Model	10	8	1	4
SSM Influence Design Pattern	9	9	1	2

A greater cyclomatic complexity has a number of ramifications. First, it shows the complexity of the program itself, usually meaning higher maintenance costs. It also shows strong interconnectivity, which makes the program less modular and more difficult to change and adapt. Finally, it also shows how much testing is required to be reasonably assured of the program's proper execution. For each of these, a smaller cyclomatic complexity is desired. It is most evidently delivered here through the use of SSM.

Qualitative Measurements. One can also measure the quality of the various models shown above with respect to some desirable parameters like the reusability factors of various models. In the Stable Architectural Pattern example, a developer can consider a total of 15 core classes along with many peripheral IOs. If one considers only the core classes, then the model is 100% reusable. As shown in table 18 below.

One can see this clearly, given the formula $RF = C_R/C_T$. Where RF is the Reusability Factor, C_R is the number of Reusable Classes, and C_T is Total Class count.

Table 18

Reusability factor for influence models.

Model	C_R	C_T	RF
Traditional Influence Model	2	8	25%
SSM Influence Design Pattern	9	9	100%

Evidently, the SSM model has a higher reusability that the traditional model, which was only given the two classes that were, in a sense, outside the scope of the program's control (money and bank account). All other objects would require significant rewriting to work in a new situation. Meanwhile, the purpose of SSM is to reuse the core classes, extending only as required, allows us 100% reusability of the core classes.

Conclusion

With the sample scenarios and related metrics given above, Stable Software Models are far more adaptable software solutions when compared to traditional modeling techniques. Though one can generate relatively flexible models, such as the one given in Figure 29 that can apply to a range of closely related scenarios, a more effective long-term solution, SSM, still remains more relevant for a variety of scenarios. In the meantime, there is necessarily some initial startup cost in implementing the core classes involved in the process. However, the core of the program may easily be reused with minimal or no changes which ultimately makes this option more attractive.

In addition, one will also find that there is some additional sharing of classes between the analysis and design patterns that are mentioned above. All of them include Actor, Party, and Entity. When these classes are implemented, they can be indexed in a pattern library to be used in all future patterns, which further increases reuse thereby reduces software development costs.

Chapter 9: Future Work and Conclusion

"Let us hear the conclusion of the whole matter…"
– Ecclesiastes 12:13

Conclusion

The utility of USER in the development of stable and reusable software systems should be easily apparent in light of the examples, scenarios, and situations shown throughout this thesis. For each concept that was studied in the last few chapters, at least two substantially different scenarios, but as many as five, were given for which a USER built solution was equally effective as a custom-built software. The difference is that the USER variant of software was reusable in other similar situations, and in many others that were quite dissimilar.

The reusability of the patterns throughout this thesis is secondary, however, to the goal, which is the description of a system for unifying the problem space of many software scenarios into only a few, which are subsequently solved through reuse of pattern-based software. The utility for such a system is obvious. Through an understanding of reuse, context, and the design and analysis patterns, near infinite reuse is not only possible but feasible.

How often must software developers and consultants labor on a software system at one organization, only to move to another and reimplement the same system from scratch due to the variations in the systems, or goals, of the new organization? It seems the answer today is: "All too often" when it should be "not at all." With USER, once a single solution has been created, all other similar solutions are trivial alterations to some peripheral components.

USER may not be popular among software engineers. There are those who prefer, or enjoy, the tedium of continuously re-implementing the same systems. It offers a measure of job security and does not necessarily require much thought to do so. However, technology is ever marching forward and waits not on the conveniences of individuals. If a new and better way emerges, the free market will move to ensure it flourishes. Ultimately, that will be the test that USER must pass to prove itself.

Future Work

There remains much to be done to prepare USER for a practical role outside of academia. New patterns must be developed, objects must be implemented, and solutions be made available to businesses and other organizations. This last aspect is primarily a business matter and can be treated as such, but development and implementation of USER will be essential parts of any further intellectual development of USER.

Pattern Development

The careful observer will note that USER is based heavily on patterns. As such, in order to ensure the success of USER, a substantial library of such patterns will need to be created. It is of little use to any organization to develop software based on a pattern that does not yet exist. Since each new pattern can be added to the library of patterns, the need for new patterns will decrease over time.

The process of creating a pattern is simple, though not necessarily easy. To create an adequate pattern, it is first necessary to determine all of the fundamental attributes, the EBTs, of the concept being worked with. With an EBT as the base, some of the more common business objects can be added almost without thought based on an abstract knowledge of the EBT involved. However, in each pattern, there are other, less common, business objects that should be included. Though there are many ways to find these, one of the simplest is to think up scenarios to determine what objects should, and should not, be a part of the pattern. The more scenarios that can be analyzed, the better the pattern should be.

Training

Though using a pattern is an almost trivial matter, some measure of practice and experience is essential to develop new patterns properly. As a part of pattern development, a small group of software engineers will need to be trained in the techniques of USER pattern design. Specific experience in software engineering itself is not necessarily essential since most of the design process is more information structuring than programming. Additionally, vast experience in older programming techniques may prove a hurdle, albeit surmountable, to learning these new techniques.

Implementation

Another vital step for making USER work to its full potential is to implement the objects in the patterns developed for USER. Any given pattern can easily use a dozen or more EBTs and business objects, and each will have to be implemented in turn. Fortunately, many of these business objects are shared across multiple patterns, allowing us to reuse these objects. There may be some additional methods and attributes that will be added to objects as new patterns require, but the increase in the total number of objects will, over time, diminish as with the development of patterns.

Even without an implementation, the USER patterns are still useful. They can help analyze situations and abstract them in an effort to find solutions to those very situations. This may be done without software, as in the case study of stress presented earlier.

Patterns can be used to present domain knowledge and facilitate abstraction, but to turn the idea into software, implementation of the objects will be required.

Testing

As with any major software system today, once the patterns have been created and implemented, they must be tested. In part, this will be testing in the traditional sense of ensuring all components function individually as desired, minimizing bugs or side effects. But the more important testing aspect will be in confirming that the patterns accurately reflect a broad range of scenarios put to them to ensure that the pattern has the adaptability and extensibility expected of a USER pattern. Fortunately, since the patterns are primarily constructed of reusable classes, there will be little issue with adding or replacing business objects in the pattern.

References

Amar, L., & Coffey, J. (2005). Measuring the benefits of software reuse. *Dr. Dobb's Journal: Software Tools For The Professional Programmer*, *30*(6), 73-76. Retrieved from http://www.drdobbs.com/measuring-the-benefits-of-software-reuse/184406111

Capiluppi, A., Stol, K. J., & Boldyreff, C. (2013). Software reuse in open source: A case study. In S. Koch (Ed.), *Open source software dynamics, processes, and applications* (pp. 151-176). Hershey, PA: IGI Global. doi:10.4018/978-1-4666-2937-0.ch008

Cline, M., & Girou, M. (2000). Enduring business theme. *Communications of the ACM*, 43 (5), 101-106. doi:10.1145/332833.332846

Cobbett, R. (2009, May 05). The most successful game ever: a history of Minesweeper. [Web log post] Retrieved from http://www.techradar.com/news/gaming/the-most-successful-game-ever-a-history-of-minesweeper-596504

Constantinou, E., Ampatzoglou, A., & Stamelos, I. (2014). Quantifying reuse in OSS: A large-scale empirical study. *International Journal of Open Source Software and Processes (IJOSSP)*, *5*(3), 1-19. doi:10.4018/IJOSSP.2014070101

Coulange, B. (2012). *Software reuse* (I. Craig, Trans.). London, England: Springer Science & Business Media. doi:10.1007/978-1-4471-1511-3

Dellarocas, C. (2003). The digitization of word of mouth: Promise and challenges of online feedback mechanisms. *Management science*, *49*(10), 1407-1424. doi:10.1287/mnsc.49.10.1407.17308

Dewe, P. J., O'Driscoll, M. P., & Cooper, C. L. (2012). Theories of psychological stress at work. In R. Gatchel & I. Schultz (Eds.), *Handbook of occupational health and wellness* (pp. 23-38). doi:10.1007/978-1-4614-4839-6_2

Ezran, M., Morisio, M., & Tully, C. J. (2002). *Practical software reuse*. London: Springer Verlag. doi:10.1007/978-1-4471-0141-3

Fayad, M.E. (2002a). Accomplishing software stability. *Communications of the ACM*, *45* (1), 111-115. doi:10.1145/502269.502308

Fayad, M.E. (2002b). How to deal with software stability. *Communications of ACM*, *45* (4), 109-112. doi:10.1145/505248.505278

Fayad, M. E. (2017). *Stable analysis patterns for software and systems.* Boca Raton, FL: Auerbach Publications.

Fayad, M. E., & Altman, A. (2001). Thinking objectively: An introduction to software stability. *Communications of the ACM*, *44*(9), 95-98. doi:10.1145/383694.383713

Fayad, M., & Cline, M. P. (1996). Aspects of software adaptability. *Communications of the ACM*, *39*(10), 58-59. doi:10.1145/236156.236170

Fayad, M. E., & Flood III, C. A. (2015). A pattern for stress and its resolution. *i-manager's Journal on Software Engineering*, *10*(1), 1-5. Retrieved from http://www.imanagerpublications.com/JournalIntroduction.aspx?journal=JournalonSoftwareEngineering

Fayad, M. E., Sanchez, H. A., Hegde, S. G., Basia, A., & Vakil, A. (2014). *Software patterns, knowledge maps, and domain analysis*. Boca Raton, FL: Auerbach Publications.

Fayad, M. E., Sánchez, H. A., & Singh, S. K. (2010). Knowledge maps – Fundamentally modular approach to software architecture, design, development and deployment. In I. Rahal & R. Zalila-Wenkstern (Eds.), *19th International Conference on Software Engineering and Data Engineering 2010 (SEDE-2010)* (pp. 127-133). Retrieved from https://www.researchgate.net/publication/215617656_Knowledge_Maps_-_Fundamentally_Modular_Approach_to_Software_Architecture_Design_Development_and_Deployment

Fayad, M. E., & Singh, S. K. (2010). Software stability model: Software product line engineering overhauled. In *Proceedings of the 2010 Workshop on Knowledge-Oriented Product Line Engineering* (Article No. 4). doi:10.1145/1964138.1964142

Fayad**,** M. E., & Wu, S. (2002). Merging multiple conventional models in one stable model. *Communications of ACM*, *45*(9), 102-106. doi:10.1145/567498.567529

Galorath, D. D. (2008). Software total ownership costs: Development is only job one. *Software Tech News*, *11*(3). Retrieved from http://www.galorath.com/DirectContent/software_total_ownership_costs-development_is_only_job_one.pdf

Griss, M. (1995). *Software reuse: Objects and frameworks are not enough* (Report HPL-95-03). Palo Alto, CA: Hewlett-Packard Laboratories, Technical Publications Department. Retrieved from https://pdfs.semanticscholar.org/584a/1480a1f4458dac856e31b978120c800f220d.pdf

Hamza, H. (2002). *A foundation for building stable analysis patterns.* (Master's thesis). Retrieved from https://pdfs.semanticscholar.org/aa99/ce14bc9e390c57cfe2d3e8cb19c9eae9277b.pdf

Hamza, H., & Fayad, M. E. (2002, September). *A pattern language for building stable analysis patterns.* Paper presented at the 9th conference on pattern languages of programs 2002 (PLoP02), Monticello, IL. Retrieved from https://www.researchgate.net/publication/215586145_A_Pattern_Language_for_Building_Stable_Analysis_Patterns

Hamza, H., & Fayad, M. E. (2002). Model-based software reuse using stable analysis patterns. Retrieved from https://www.researchgate.net/profile/Mohamed_Fayad/publication/228919069_Model-based_software_reuse_using_stable_analysis_patterns/links/0f317538d3f52eeb03000000/Model-based-software-reuse-using-stable-analysis-patterns.pdf

Hamza, H., Mahdy, A., Fayad, M. E., & Cline, M. (2003). Extracting domain-specific and domain-independent patterns. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 310-311). doi:10.1145/949344.949427

Kan, S. H. (2002). *Metrics and models in software quality engineering*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Koch, C. (1996). The bright stuff. *CIO*, *9*(11), 56-62.

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys (CSUR)*, *24*(2), 131-183. doi:10.1145/130844.130856

Kulkarni, N., & Varma, V. (2016). Perils of opportunistically reusing software module. *Software: Practice and Experience*. Advance online publication. doi:10.1002/spe.2439

Mahdy, A., & Fayad, M. E. (2002). *A software stability model pattern.* Retrieved from https://www.researchgate.net/publication/2559402_A_Software_Stability_Model_Pattern

Mahdy, A., Fayad, M. E., Hamza, H., & Tugnawat, P. (2002). Stable and reusable model-based architectures. In *12th Workshop on Model-based Software Reuse* (Vol. 16). Retrieved from https://www.researchgate.net/profile/Mohamed_Fayad/publication/215585444_Stable_and_Reusable_Model-Based_Architectures/links/54757eec0cf245eb4370c882.pdf

Mili, H., Mili, A., Yacoub, S., & Addy, E. (2001). *Reuse based software engineering: Techniques, organization and measurement*. New York: Wiley.

Mojica, I. J., Adams, B., Nagappan, M., Dienst, S., Berger, T., & Hassan, A. E. (2014). A large-scale empirical study on software reuse in mobile apps. *IEEE Software, 31*(2), 78-86. doi:10.1109/MS.2013.142

Morisio, M., Ezran, M., & Tully, C. (2002). Success and failure factors in software reuse. *IEEE Transactions on software engineering*, *28*(4), 340-357. doi:10.1109/TSE.2002.995420

Myers, W. (1978). The need for software engineering. *Computer*, *11*(2), 12-26. doi:10.1109/C-M.1978.218054

Nurolahzade, M., Walker, R. J., & Maurer, F. (2013, June). An assessment of test-driven reuse: Promises and pitfalls. In *International Conference on Software Reuse* (pp. 65-80). doi:10.1007/978-3-642-38977-1_5

Riehle, D., & Züllighoven, H. (1996). Understanding and using patterns in software development. *Theory and Practice of Object Systems, 2*(1), 3-13 doi:10.1002/(SICI)1096-9942(1996)2:1<3::AID-TAPO1>3.0.CO;2-#

Savitz, E. (2012, April 16). The big cost of big data. *Forbes.* Retrieved from http://www.forbes.com

Schmidt, D. C. (1999). Why software reuse has failed and how to make it work for you. *C++ Report*, *11*(1).

SEER for Software - Estimating Software Development & Maintenance Costs (Version 7.3) [Computer Software]. El Segundo, CA: Galorath, Inc.

Teichroew, D., Hershey, E. A., & Yamamoto, Y. (1978). Computer-aided software development (CERN Report 78-13). Retrieved from http://www.iaea.org/inis/collection/NCLCollectionStore/_Public/10/462/10462028.pdf?r=1

Tamai, T., & Monpratarnchai, S. (2014, December). A context-role based modeling framework for engineering adaptive software systems. In *2014 21st Asia-Pacific Software Engineering Conference,* 1, 103-110. doi:10.1109/APSEC.2014.25

Tracz, W. J. (1988). Software reuse myths. *ACM Software Engineering Notes, 13*(1), 17-21. doi:10.1145/43857.43859

Walters, G. F., & McCall, J. A. (1979). Software quality metrics for life-cycle cost-reduction. *IEEE Transactions on Reliability*, *28*(3), 212-220. doi:10.1109/TR.1979.5220569

Wu, S., Hamza, H., & Fayad, M. E. (2003). Implementing pattern languages using stability concepts. Paper presented at the meeting of ChiliPLoP 03', Carefree, AZ. Retrieved from https://pdfs.semanticscholar.org/bfb3/79cafc09c4462c32fb99077cbf22e5afce5c.pdf?_ga=2.82355285.252115093.1497758123-2053123364.1497758123

Appendix A: Knowledge Map Template for Software Reuse

Software reuse or code reuse is the use of existing software, or software knowledge, to build new software following the reusability principles. A reusable component may be code, but the bigger benefits of reuse come from a broader and higher-level view of what can be reused. Software specifications, designs, tests cases, data, prototypes, plans, documentation, frameworks, and templates are all candidates for reuse.

Software reuse is a major component of many software productivity improvement efforts, because reuse can result in higher quality software at a lower cost and delivered within a shorter time. Reuse takes place when an existing artifact is utilized to facilitate the development or maintenance of the target product. The scope of reuse can vary from the narrowest possible range, namely, from one product version to another, to a wider range such as between two different products within the same line of products or between products in different product lines. The scope of reuse is limited, in general, by the nature of and constraints on a product line; for example, it is unwise to reuse a desktop application in a mission-critical system.

Tables A1 and A2 below list and describe the goals (EBTs) and capabilities (BOs) requisite for software reuse. Table A3 maps select BOs to EBTs demonstrating which are correlated.

Table A1

EBTs of "Software Reuse"

EBTs/goals	Description
Reuse	To reuse something is to use it again after it has been used once. This includes conventional reuse in which the item is used again for the same function and creative reuse in which it is used for a different function.
Abstraction	Abstraction is the act or process of separation. It is that particular view of a problem that extracts information relevant to a purpose and ignores the rest. Abstraction is one of the convenient ways to deal with complexity.
Unification	Software unification is about bringing together all IT application assets under a single, elastic, and location-transparent abstraction layer whereby functions from all existing heterogeneous applications and system are available in a single environment as if they belonged to a single application.
Modularity	Modularity is the act of separating the functionalities of a program such that each of these modules is independent and is sufficient to execute only one aspect of the program.
Stability	Stable software is so named because it is unchanging. Its behavior, functionality, specification or API is considered 'final' for that version. Apart from security patches and bug fixes, the software will not change for as long as that version of the software is supported, usually from 1 to many years.
Risk management	Risk management involves reducing the probability of occurrence of all uncertain events and also helps to measure the loss that they would cause.
Standardizing	A software standardizing process involves a standard, protocol, or other common format of a document, file, or data transfer accepted and used by one or more software developers while working on one or more than one computer programs. These standards enable interoperability between different programs created by different developers.

EBTs/goals	Description
Development	Software development is the process of programming, documenting, testing and bug-fixing various applications and frameworks in order to build a software product.
Economizing	Economizing is the process of managing and essentially reducing the overall cost involved in a process. Software reuse makes a process more cost effective by reducing need for more resources.
Optimizing	Optimizing is the process of generating the most favorable results in least duration of time. The reuse of software reduces the development time involved to a great extent and generates a highly optimized product.
Depending	Software is said to be depending in nature if it has availability, reliability, and maintainability which may also encompass the mechanisms designed to increase and maintain the dependability of a system.

Table A2

Business Object Requisite for the Software Reuse Pattern

BOs/Capabilities	Description
Party	A group of people organized together to further a common aim or taking part in a particular activity. In software reuse, parties are the people who use some existing resource to create new software and are further involved in its reuse implementation.
Reason	A cause, explanation, or justification for an action or event. The cause of using existing resource in the design is driven by the fact that it leads to reduction in effort, time and cost of software production by replacing creation with recycling. Reusing of proven legacy software also ensures usage of bug-free components to build new software.
Outcome	An end result or a consequence from an action. The final software that includes the reusable parts is highly dependable software which adheres to the standard compliances.
Mechanism	A natural or established process by which something takes place or is brought about. The technique for software reuse depends on an architecture driven approach to software development.
Medium	An intervening substance or agency for transmitting or producing an effect. The platform that the existing resource and final software play on.
Function	A basic task of a computer, especially one that corresponds to instructions from the user. The function that the party need and the existing resource can implement can vary from one party to another and accordingly, the functionalities can be implemented in a software for the parties separately.

BOs/Capabilities	Description
Asset	An asset can be defined as a useful or valuable quality or thing; an advantage or resource. The key advantages of reusing assets are: • Increase delivery efficiency • Innovate from a higher level • Use best practices • Reduce risk • Offer wider array of flexible solutions
Context	Context signifies the set of circumstances that surround a particular task undertaken. Software reuse depends on the context in which it is implemented and thus, one must follow a systematic approach toward it.
Frequency	Frequency is the state of being frequent or occurring often. Frequency with which engineers are using a software is an important metric for defining its reusability.
Level	One of the promises of object-orientation is reuse. Developing new software systems is expensive, and maintaining them is more expensive. Reuse is therefore sensible in both business and technology perspectives. One must carefully identify the levels in software reuse.
Pattern	It is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.
Framework	Developers often lack knowledge of, and experience with, fundamental design patterns in their domain, which makes it hard for them to understand how to create and/or reuse frameworks and components effectively.

BOs/Capabilities	Description
Use Cases	We incorporate reuse components in the initial phases of the software development process, that is to say, requirements specifications. These components, use case in the pattern form, reused during the requirements capture, allow a visualization of the system to be implemented.
Test Cases	Reusable test suites should only be created under the appropriate circumstances. These test cases can be later on reused in the future uses of the software. Such test cases have to be created carefully.
Methods	Well-defined methods are of utmost importance to achieve reusability in software. These are the principles or guidelines which give a direction to the entire process of software reuse. Thus, methods play a vital role.
Architecture	Software architecture refers to the high level structures of a software system, the discipline of creating such structures, and the documentation of these structures. These structures are needed to reason about the software system. These are further used when on reuse.
Polymorphism	It is the ability to Having, or assuming, various forms, characters, or styles. One of the major goals of OOP is software reuse. One can illustrate this by considering two different approaches to reuse: Inheritance -- the "*is-a*" relationship. Composition -- the "*has-a*" relationship.
Constraint	A limit or restriction of using the existing resource in the design. There can be various factors restricting the reuse of software which can range from lack of tool support to involved maintenance costs.
Contract	Software component is associated with a contract which gives the formal model of its functional behavior. Software is administered, retrieved and reused by its contract. The reuse of software is determined by how its contract is structured.

BOs/Capabilities	Description
Defect	Defect is a shortcoming or imperfection in software which puts limitations on its working or functionalities. Software reuse improves quality of software leading to reduction in defect density in it.
Risk	It is the probability of occurrence of uncertain events and their potential leading to a loss for an organization using that software. The reuse of software reduces the margin of error in its cost estimation in a project and also leads to reduced process risks.
Scope	Scope assesses or investigates something to give its range of view. The scope of Software reuse can be divided into product reuse and process reuse.
Resource	Resource defines the materials, strategies etc. undertaken for the completion of a task. Software reuse leads to the saving of resources to a great extent. Reusable resources can be template, component, framework, artifact, pattern apart from the reuse of code and inheritance.
Project	Project outlines a detailed plan to accomplish a task involving considerable goals, resources, cost and personnel. Reuse between projects is where one can think of taking the greatest advantage.
Function	A basic task of a computer, especially one that corresponds to instructions from the user. The function that the party need and the existing resource can implement can vary from one party to another and accordingly, the functionalities can be implemented in a software for the parties separately.
Scenario	A scenario is an outline of a subsystem. It includes a sequence of possible events to be studied in a system of interest.
Model	An abstract system, obeying certain specified conditions, which purpose is to study or illustrate an entity or event.
Library	A collection of standard programs and subroutines for immediate use

BOs/Capabilities	Description
Component	One element of a large system, which purpose is to implement a specific function.
Object	A self-contained module of data and its associated processing. Objects are the software building blocks of object technology.
Diagram	A schematic representation of a sequence of subroutines designed to solve a problem
Class	A description of the structure and operations of an object. Any one of this collection share the same characteristics.
Layer	Architects should look to reuse significant application frameworks such as layers that can be applied to many different types of applications.

Table A3

Knowledge Map of Software Reuse showing requisite BOs for given EBTs

EBTs	BOs
Risk management	Party, Reason, Outcome, Mechanism, Medium, Function, Constraint, Contract, Defect, Risk, Scope, Resource, Context
Standardizing	Party, Reason, Outcome, Mechanism, Medium, Function, Constraint, Contract, Defect, Risk, Scope, Resource, Project, Context
Development	Party, Reason, Outcome, Mechanism, Medium, Function, Constraint, Contract, Defect, Risk, Scope, Resource, Project, Context
Economizing	Party, Reason, Mechanism, Medium, Resource, Defect, Project, Constraint
Optimizing	Party, Reason, Mechanism, Medium, Resource, Function, Project, Constraint
Depending	Party, Reason, Outcome, Mechanism, Medium, Function, Constraint, Contract, Risk, Scope, Resource
Reuse	Party, Mechanism, Type, Entity, Event, Criteria, Artifact, Media, Evidence
Abstraction	Party, Type, Criteria, Evidence, Entity, Event, Media, Log
Ownership	Party, Actor, Mechanism, Context, Criteria, Log, Type, Entity, Event, Media
Encapsulation	Party, Type, Scenario, Criteria, Entity, Event, Application, Media

Appendix B: Knowledge Map of CBSD

Context Based Software Development characterizes the work environment of a software developer by considering all dimensions. The term context here implies various circumstances around an area of interest. Useful contextual information can also be extracted by using the project management tools.

Tables B1 and B2 below list and describe the goals (EBTs) and capabilities (BOs) utilized in Context Based Software Development. Table A3 maps select BOs to EBTs demonstrating which are correlated.

Table B1

EBTs of Context Based Software Development

EBTs/Goals	Description
Simplicity	Context information needs to represent information ranging from being simple. The simplest structures are easier to build and maintain.
Unification	All dimensions of context information provided by a working environment of a developer needs to be integrated for creating a unified context model.
Reusability	Software components needs to be reused by focusing on the information captured during the process development.
Adaptability	An ability to change something to fit to occurring changes, or cope with unexpected changes.
Configuration	Software configuration is the task of making tracking of changes in software and controlling them.
Applicability	Applicability represents how the context model is used and the objectives behind its use.
Extensibility	All the remaining layers of context model will be referenced iteratively, as an extent to the developed work already.
Personalization	For a developer working on a specific task, needs to be dealing with various resources for accomplishing the task.
Customization	Information retrieval facilities are to be customized in order to support the reuse of software components.
Abstraction	There are various layers in context model that are hidden/abstracted from a user to handle directly.

Table B2

BOs of Context Based Software Development

BOs/Capabilities	Description
Context	Network of elements across different dimensions that are not limited to work developed on IDE.
Scenario	It is where most of the actions take place and where large amounts of contextual information is available.
Application	It refers to the use of Context model and objectives behind its use.
Pattern	Dotted and filled patterns represent explicit and implicit relations respectively. Common patterns are used to perform search operations.
Design	Design includes a layered model requiring a typical developer to focus on different layers in a working environment.
Architecture	The developer context model has the layered architecture and requires integrating all the dimensions.

Table B3

Knowledge Map of CBSD

EBTs	BOs
Simplicity	Party, Mechanism, Entity, Event, Criteria, Outcome
Unification	Party, Criteria, Outcome, Context, Scenario, Mechanism, Entity, Event
Reusability	Outcome, Entity, Event, Media, Mechanism, Party, Constraint, Criteria
Adaptability	Entity, Event, Impact, Consequence, Party, Cause, Media, Mechanism
Configuration	Impact, Outcome, Criteria, Constraint, Mechanism, Party, Type
Applicability	Context, Type, Media, Party, Actor, Mechanism, Criteria, Entity, Event, Rule, Form
Extensibility	Party, Entity, Event, Mechanism, Criteria, Outcome, Level
Personalization	Party, Actor, Mechanism, Entity, Event, Outcome, Impact, Criteria
Customization	Party, Actor, Mechanism, Entity, Event, Outcome, Impact, Criteria
Abstraction	Party, Criteria, Entity, Event, Mechanism, Level, Impact, Outcome, Media