DRAFT

Software Reuse Knowledge Map

Mohamed E. Fayad, PhD[1] and Charles A. Flood III[2]
Department of Computer Engineering,
Charles W. Davidson College of Engineering,
San Jose State University, San Jose, CA, USA
[1]m.fayad@sjsu.com and [2]charlesFlood3@gmail.com

The purpose - where I start - is the idea of use. It is not recycling, it's reuse.
Issey Miyake [1]

The term Software Reuse can be defined as a process of developing software systems from existing ones, instead of creating them from scratch [2, 3, 4, 5]. In many software engineering disciplines, systems are designed by recomposing existing software components that have been used in other systems. It has become a topic of renewed interest in the software community because of its potential benefits, which include increased product quality, decreased product cost and schedule. Software is rarely built completely from scratch. To a great extent, existing software documents (source code, design documents, etc.) are copied and adapted to fit new requirements. Software engineering is focused more on original development. However, it is now recognized that to create better software in a time efficient and cost effective way, we may need to adopt a design process that is based on systematic software reuse factors.

This paper aims at applying Software Stability Model (SSM) approach towards creating a model for *Unified Software Reuse* that is applicable in all scenarios and diverse domains. The software stability notion ensures high reuse ability, stability and a more design efficient model, which is domain independent too. The key contribution of this Paper is to present an idea of having a stable analysis pattern listing the Enduring business themes (EBTs) and Business Objectives (BOs) involved in the area of Unified Software Reuse. Such a generic model can further be applied to any possible scenario.

3.1 Introduction:

Traditional Software Reuse:

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

Demand for more complex and technically evolved software applications with greater and efficient software content is growing at a significant rate. Of late, software market has witnessed diverse varieties of applications that cater to an equally diverse number of industries and businesses. Hence, market cycles for some software products in terms of market windows and product cycles have decreased at an alarming rate. However, software production methods are

DRAFT

not evolving at a similar rate while user demand for better and highly equipped systems are forcing developers to think of creating newer designing technologies that can turn out better software applications at faster cycling rates. Software developers also feel the need for improved time-to-market rates, better quality and enhanced productivity in their daily operations. Cost of development, efforts and budget are the other important factors that are forcing them to look for more innovative methods that can significantly improve the designing process of software applications. Although different solutions have been proposed and followed, most of them have been following a single solution approach that seriously hinders productivity cycles.

One of the suggested software designing methods is software reuse method. This simple, yet powerful vision was introduced in 1968. Reuse of software is based on a simple, well-known idea. When we build a new firmware or software application, we should reuse previously developed software components. Doing so will save the cost of developing, testing, documenting and maintaining multiple copies of essentially similar software. Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used. Reuse is possible at a range of levels from simple functions to complete application systems. A traditional view of reusable software system approach is rooted in the creation of software libraries that contain generic and reusable components that could be gelled together to design new software systems. Often, this is forms the base of traditional reusability research. In effect, reusable software research utilize existing reusable resources that are considered atomic building blocks that are eventually indexed, organized and combined by using well-defined rules and regulations.

The traditional approach of creating a new software application follows an approach that requires a considerable quantum of new codes written over a period. However, overall cost and time for development and its overall quality depends entirely on the volume of codes composed by a developer. The fallback method to prevent such an occurrence is to write lesser quantum of codes and reduce the time and money required to create a new software application. It may make sense to gather and accumulate available software components from a library and reuse them to write a new application. As developers, increase the shelf of newer software products by using an already existing library of codec, they can easily improve cost, time and quality parameters. Hence, reuse approach to create software products is a well-devised strategy for developers, who can follow the current market trends that demand technical products as a faster turn-around rates.

DRAFT

Abstraction plays a central role in software reuse. Concise and expressive abstractions are essential, if software artifacts are to be effectively reused. Software reuse is the use of existing assets in some form within the software product development process. More than just code, assets are products and by-products of the software development life cycle and include software components, test suites, designs and documentation. Leverage is modifying existing assets as needed to meet specific system requirements. Because reuse implies the creation of a separately maintained version of the assets, it is preferred over leverage. 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.

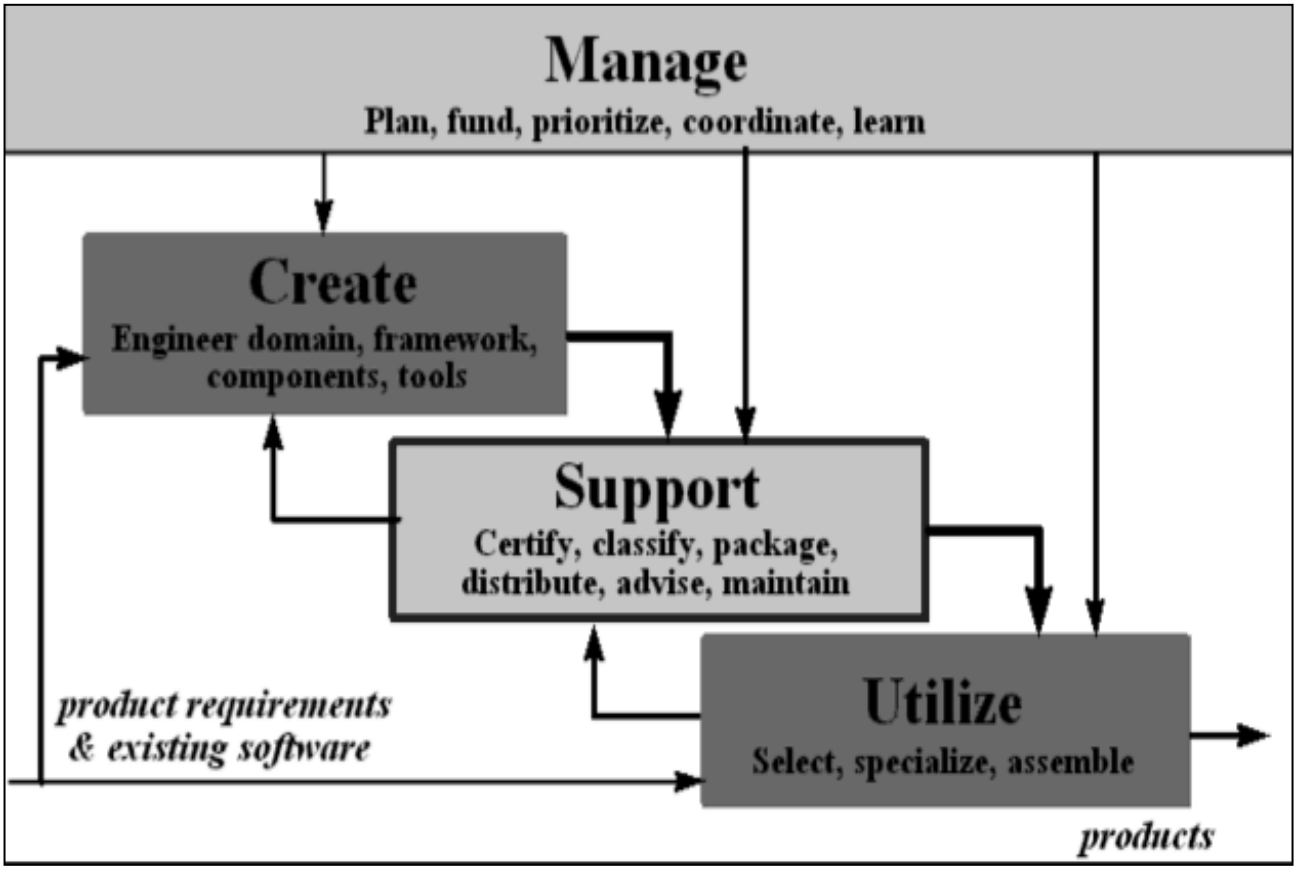


Figure 3.1: Basic-Software reuse framework

"Waste not, want not" has never been the motto of software developers. On the contrary, waste has been encouraged as a normal part of the one-of-a-kind system development philosophy. The need for 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

DRAFT
specifically designed to meet a set of particular customer requirements, no matter how much reinventing-of-the-wheel occurs. Software Reuse can happen in various ways like:

1. Application System Reuse
   - reusing an entire application by incorporation of one application inside another (COTS reuse)
   - development of application families (e.g. MS Office)
2. Component Reuse
   - components (e.g. subsystems or single objects) of one application reused in another application
3. Function Reuse
   - reusing software components that implement a single well-defined function

3.2 Pitfalls of Traditional Software Reuse:

Challenges/Constraints in software reuse: There are many challenges that developers still need to overcome, when to the issue of implementing the software reuse strategy arises. Some of these challenges are [6, 7, 8, 9, 10]:

- Increased maintenance costs: If the source code of a reused software system or component is not readily available, then maintenance costs might increase, as the reused elements of the system will become increasingly incompatible with any system changes.
- Lack of tool support: CASE toolsets may not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account.
- Not-invented-here syndrome: Some software engineers may prefer re-writing components, as they believe that they can improve on the reusable component. This is partly to do with trust and to some extent with the fact that writing original software is seen as more challenging than reusing other people's software.
- Creating and maintaining a component library: Populating a reusable component library and ensuring the software developers can use this library could be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature so to say.
- Finding, understanding and adapting reusable components: Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library, before they will routinely include a component search as part of their normal development process.

3.3 Essential Properties of Software Reuse

These benefits of Unified Software Enginnering Reuse can be quite significant. Some of them are listed below:

DRAFT

- **Reuse of programs** within AT&T, GTE, Ericsson, NEC, Toshiba, and HP itself, has demonstrated that time to market can be improved by a factor of 1.5 to 2.5 or more. – Unverified claim
- **Software or firmware** that used to take 24 months or longer to produce, can take only 6 months or less to develop with reuse. – Unverified claim
- **Quality can be increased by 5-10 times**, and costs of development and maintenance can be reduced. – Unverified claim
- **Reuse of standard modules** can increase the interoperability and consistency of products.
- Appropriate components and supporting tools can enable a professional services organization to rapidly produce ``almost custom solutions" cost effectively. – Unverified claim
- Increase software productivity -- – Unverified claim
- Shorten software development time – Unverified claim
- Improve software system interoperability – Unverified claim
- Develop software with fewer people
- Move personnel more easily from project to project
- Reduce software development and maintenance costs
- Produce more standardized software
- Produce better quality software and provide a powerful competitive advantage

3.4 Overview of Software Reuse Knowledge Map:

Our knowledge map consists of following two EBTs and Bos [11, 12,13, 14].

1. EBTs: These are "Enduring Business Themes" which represent the elements that remain stable internally and externally.

a) REUSE: To reuse something is to use it again after it has been used once. This includes conventional reuse, where the item is used again for the same function, and creative reuse, where it is used for a different function. Build frameworks that are modular and can be easily reused. Test automation, although very valuable, is often a very expensive effort, where the ROI (return on investment) becomes questionable.This is primarily because of changing product functionality, which may invalidate the test scenario at hand. While 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. By poor test automation strategy and code, we mean choice of cases that don't add a lot of value when automated, huge chunks of repetitive code that is written, code that gets very cumbersome to read through, review and maintain. An answer to all this, is to modularize test automation code and create frameworks to handle repetitive functionality. For e.g. code to sign in/sign out could be easily separated and handled in a separate module to be reused as and when required.

DRAFT

b) ABSTRACTION: Abstraction is the act or process of separation. It is a 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.

2. BOs: These are the Business objects that remain internally adaptable, but externally stable.

a) ASSET: An asset can be defined as anything of material value or usefulness. There are several types of assets:

- **Best practices:** Techniques or methodologies that, through experience and research, have proven to reliably lead to a desired result.
- **Designs:** Written documents that describe a general solution to a design problem that recurs repeatedly in many projects.
- **Tools:** Set of assets that aid in development or recreation of a project, application or task.
- **Components:** Identifiable parts of a larger program or construction.

The following are some examples of reusable software assets:

- Architectural frameworks
- Architectural mechanisms
- Architectural decisions
- Constraints
- Applications

b) 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, we have to follow a systematic approach towards it.

Table 1.1: Knowledge Map of "**SOFTWARE REUSE**"

EBT	BO	PATTERN-BO
Reuse	Reusability	AnyParty, AnyMechanism, AnyType, AnyEntity, AnyEvent, AnyCriteria, AnyArtifact, AnyMedia, AnyEvidence
Abstraction	Identification	AnyParty, AnyType, AnyCriteria, AnyEvidence, AnyEntity, AnyEvent, AnyMedia, AnyLog
Ownership	Asset	AnyParty, AnyActor, AnyMechanism, AnyContext, AnyCriteria, AnyLog, AnyType, AnyEntity, AnyEvent, AnyMedia

DRAFT

Encapsulation	Context	AnyParty, AnyType, AnyScenerio, AnyCriteria, AnyEntity, AnyEvent, AnyApplication, AnyMedia

III) FIRST MID-SIZE PATTERN : ASSET STABLE DESIGN PATTERN [16]

EBT&BO:
Asset is a BO and the EBT/goal is Ownership.

Asset is a reusable product or by-product of the software development. Typical examples are code, design, specifications, user documentation, test plans, estimates, etc. The individuals or groups that create assets with the explicit purpose of reuse in mind are called producers. The users of these assets are referred to as consumers. When a producer makes an asset explicitly available for reuse, for instance, by placing it in a reuse library, the asset is said to be published.
Example: The World Wide Web (WWW) has been successfully used by producers 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 to this virtual database. Also, standards for packaging assets have emerged making reuse easier. The following diagram summarizes the types, or categories, of reuse available. The left-hand arrow indicates the relative effectiveness of each category – pattern reuse is generally more productive than artifact and framework reuse for example. Similarly, the right hand arrow indicates the relative difficulty of succeeding at each type of reuse. Code and template reuse are relatively easy to achieve because one simply needs to find the asset and work with it.

DRAFT

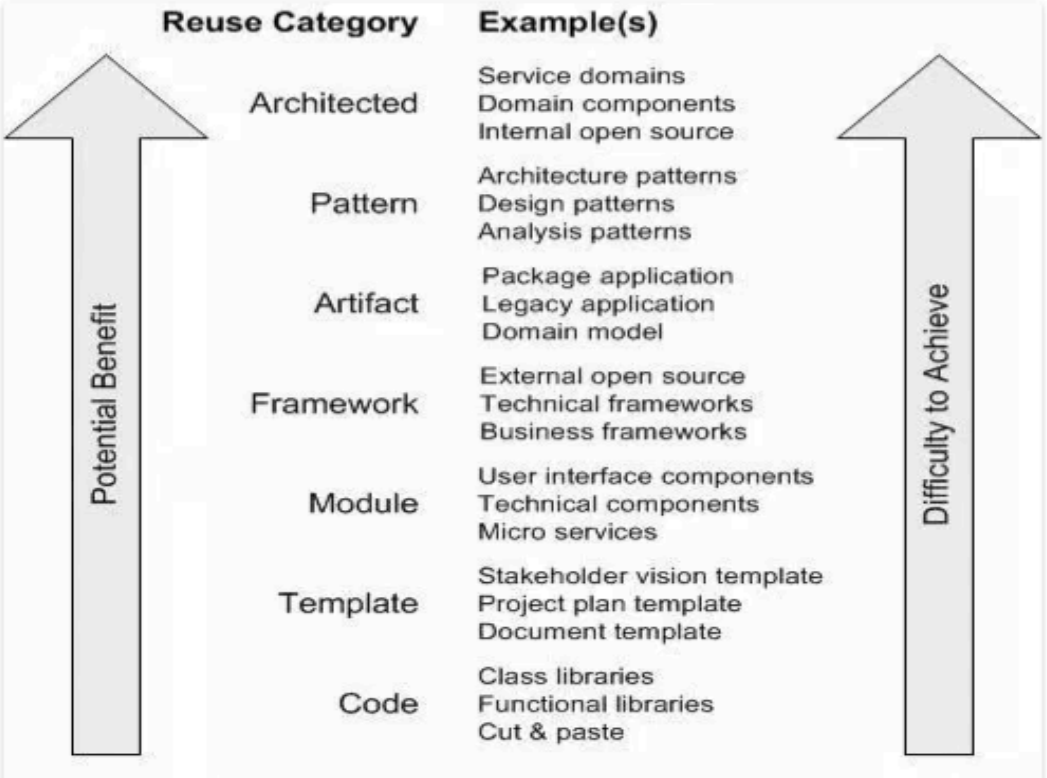


Figure 3.1: Assets that can be reused in software reuse

3.1 Context:

Assets are kind of resource that are owned by specific party. 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 a party's ownership and the evidence records in some log. The log, entity and event are on some media. Thus, the context for an asset restricts to the following functions:

AnyParty, AnyActor, AnyMechanism, AnyContext, AnyCriteria, AnyLog, AnyType, AnyEntity, AnyEvent, AnyMedia Problem

3.2 Requirements:

In the software reuse, assets have the following requirements:

1) Non-Functional requirements-
   a) **Measurable:**

DRAFT

The asset can be measured. An **asset** is an item of value owned or controlled by the entity, acquired at a measurable cost. The total book value of an entity's assets is one side of the so-called balance sheet equation:

Assets = Liabilities + Owners Equities

Assets are resources the entity owns or controls, to work with to earn income
Liabilities are what the entity owes.
Equities are what the entity owns outright.

For example, the deposit has an amount and the house has a valuation.

b) Documentable:

The asset can be recorded. For example, the deposit has transaction history and the house has title deed. Tangible asset can be identified as physical entities and are thus, clearly documentable.

c) Usable:

The asset can be used. For example, the money in the deposit account can be spent and the house can be mortgaged.

2) Functional requirements-

a) AnyParty: A party, like a person, a country or an organization can only own the asset.
b) AnyCriteria: AnyParty has some criteria to evaluate the asset; so, there is AnyCriteria in the diagram.
c) AnyEvidence: when AnyParty say that their own some assets, they need some evidence to demonstrate it; so, there is AnyEvidence in the diagram.
d) AnyLog: the evidence needs to be recorded in some place as a log; so, there is AnyLog in the diagram.
e) AnyType: the aseet has many kinds, so there is AnyType in the diagram.
f) AnyEntity and AnyEvent: the asset has many specific instances; so, there are AnyEntity and AnyEvent in the diagram.
g) AnyMedia: the log needs a media to store; so, there is AnyMedia in the diagram.

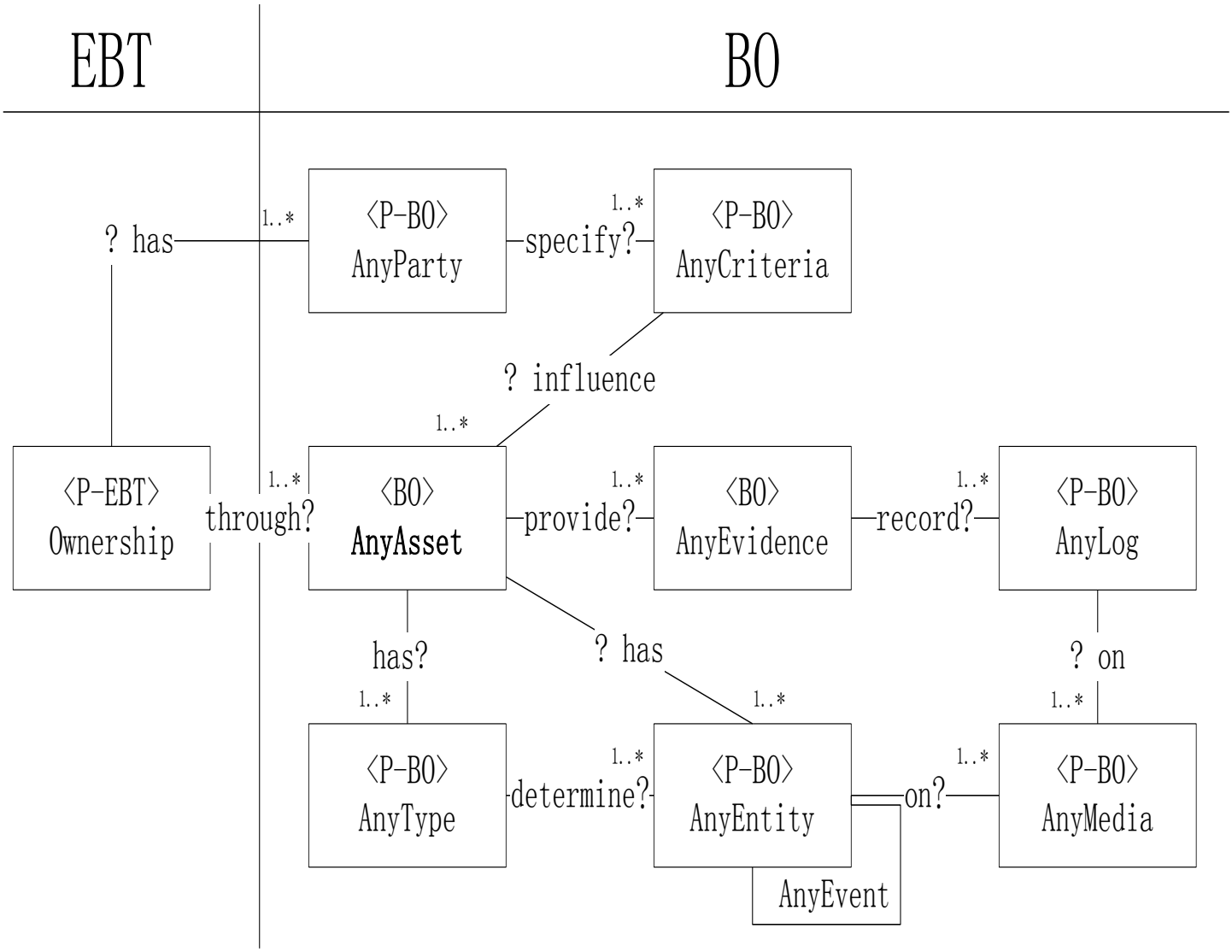


Figure 3.2: Asset Stable Analysis Pattern

Description:

1. **Ownership:** the goal of any asset is ownership.
2. **AnyParty:** the party, like a person, a country or an organization, can only own the asset. Software or hardware cannot own any asset; so, there is no AnyActor.
3. **AnyCriteria**: AnyParty has some criteria to evaluate the asset; so, there is AnyCriteria in the diagram.
4. **AnyEvidence**: when AnyParty say that their own some assets, they will need some evidence to demonstrate it; so there is AnyEvidence in the diagram.

DRAFT

5. **AnyLog**: the evidence needs to be recorded in some place as a log; so, there is AnyLog in the diagram.
6. **AnyType**: the asset has many kinds; so, there is AnyType in the diagram.
7. **AnyEntity & AnyEvent**: the asset has many specific instances: so, there are AnyEntity and AnyEvent in the diagram.
8. **AnyMedia**: the log needs a media to store; so, there is AnyMedia in the diagram.

3.3 Application

1) Application #1: Client owns some deposit in the bank

In this scenario, the **deposit (AnyAsset)** is saved in the **bank (AnyParty)**. The **client (AnyParty)** saves the deposit and the bank manages the deposit. In addition, the bank has some specific **criteria (AnyCriteria)** to describe the deposit. When the client wants to prove that they **own (Ownership), a specific type (AnyType)** of **deposit (AnyEntity)**, the bank will provide some **certifications (AnyEvidence)** to show the state and other related information about the deposit. At last, all the **transactions (AnyLog)** can be found in the **bank system (AnyMedia)**.

Table 1.2 Asset Applications #1

EBT	BOs	APP-1
Ownership	----------	----------
	AnyAsset	Deposit
	AnyParty	Client, Bank
	AnyCriteria	SpecificCriteria
	AnyType	SpecificType
	AnyEvidence	SystemLog
	AnyEntity	SpecificDeposite
	AnyLog	Certification
	AnyMedia	BankSystem

Thus, an important philosophy for succeeding at reuse in the information technology (IT) space is to understand that users have one or more options at their disposal.

DRAFT

3.4 Class Diagram:

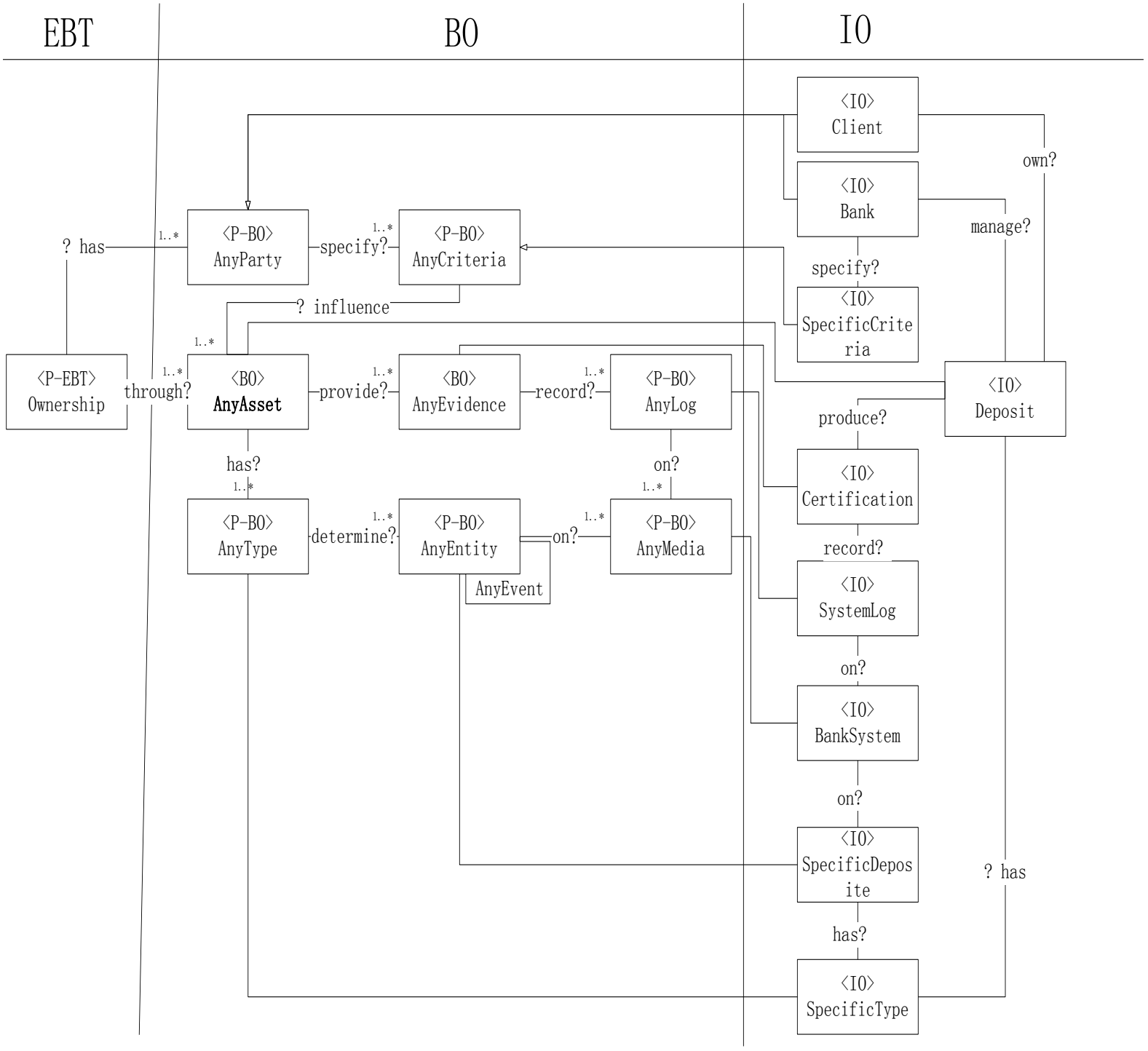


Figure 3.3: Asset Application #1

DRAFT

Use Case #1: demonstrate the ownership of a specific deposit

Table 1.3: Actor and Role

Actor	Role
Person	Client
Person	Bank Clerk

Table 1.4: Class

Class	Type	Attribute	Operation
Client	Person	id name gender	getCertification()
BankClerk	Person	id name gender	giveCertification() writeLog()
SystemLog	System Class	id size data	save() load()
Certification	System Class	id name owner	getOwner() getAsset()
Deposit	System Class	id amount type	getAmount() getType()
SpecificCriteria	System Class	id name	getdetail()
SpecificType	System Class	Name Id	getName()
SpecificDeposite	System Class	Id Name CreateTime	getOwner() getAmount() getType()
BankSystem	System Class	Name Platform	getVersion()

UC description:

1) The **AnyParty(Client)** has the **Ownership** of the **AnyAsset (Deposit)**.
2) The **AnyAsset (Deposit)** has **AnyType (SpecificType)**.

DRAFT

3) The **AnyType (SpecificType)** determines the **AnyEntity (SpecificDeposit)**.
4) The **AnyEntity (SpecificDeposit)** is stored by **AnyParty (Bank)**.
5) The**AnyParty (Client)** asks **AnyParty(Bank)** to getthe **AnyEvdence (Certification).**
6) The **AnyParty (Bank)** follows the **AnyCriteria (SpecificCriteria)** to record the **AnyLog (SystemLog)** on the **AnyMedia (BankSystem)**.
7) The **AnyParty (Bank)** returns the **AnyEvidence (Certification)** to the **AnyParty (Client)**.

3.5 Sequence Diagram:

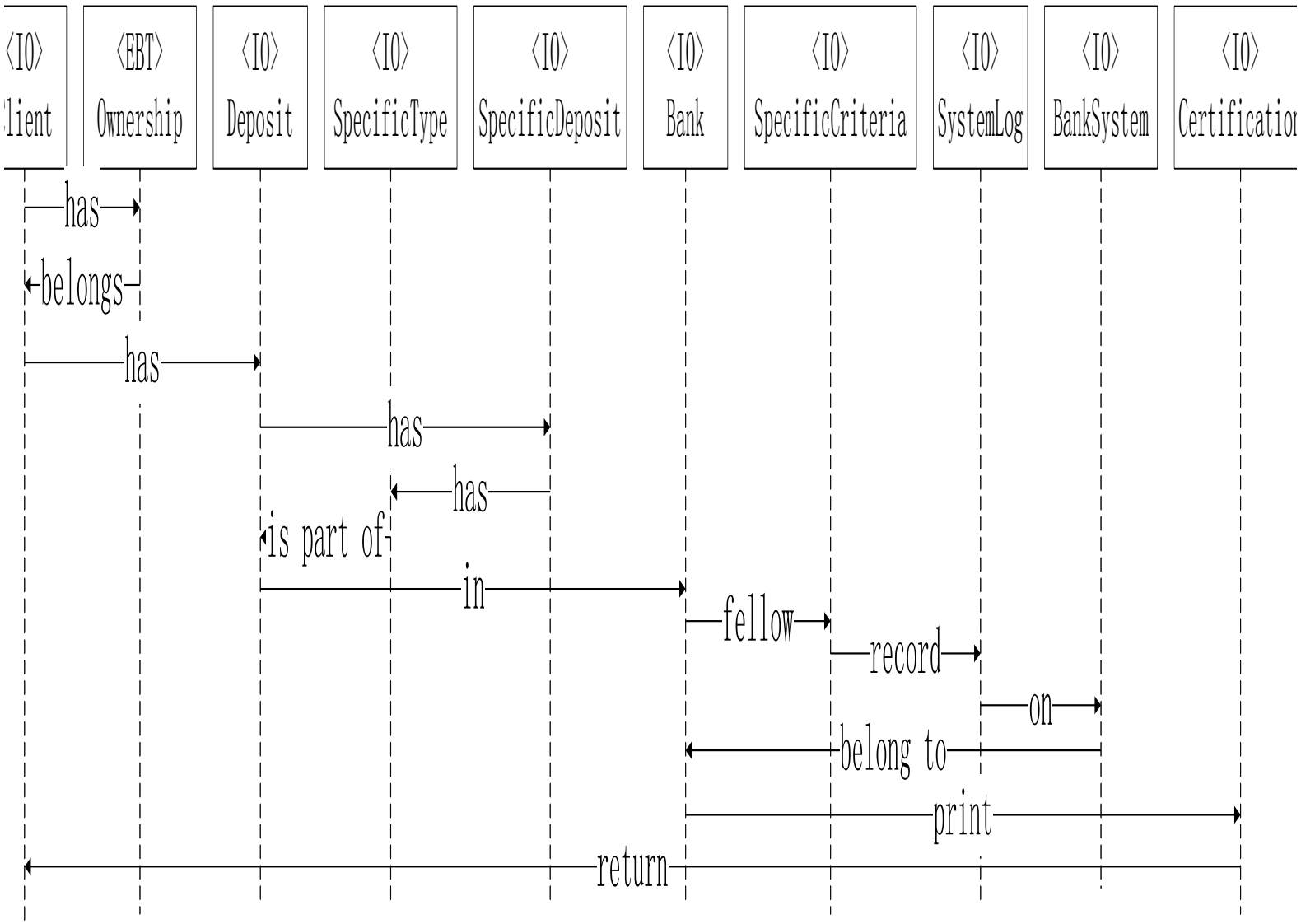


Figure 3.4: SequenceDiagram

DRAFT

IV) SECOND MID SIZE PATTERN : REUSE STABLE DESIGN PATTERN

EBT&BO

Reuse is an EBT and its BO is Reusability [15].

Software reusability is considered a way to solve a software development crisis. When we want to solve a problem, we will try to apply the solution to similar problems, because that makes our work easy and simple. Software reusability can 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 and decreased product cost and schedule. The most substantial benefits derive from a product line approach, where 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.

DRAFT

Figure 4.1: Reuse as depicted in the stages of Reuse Engineering

4.1 Context:

Any party does the reuse of an entity. We need a mechanism to carry on reuse process and an artifact will be reused repeatedly.

Thus, following operations are needed:

AnyParty, AnyMechanism, AnyType, AnyEntity, AnyEvent, AnyCriteria, AnyArtifact, AnyMedia, AnyEvidence

4.2 Requirements:

In the software reuse, reuse has the following requirements:

1. Functional requirements

a) AnyArtifact

DRAFT

A reusable executable unit is required for the software reuse

b) AnyMechanism

Mechanism is 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.

c) AnyContext

Software reuse always happens in a given context that defines the encapsulation in which we perform software reuse

2. Non-Functional requirements

a) Complete

Software reuse has to be complete in nature. Its completion is determined by the fact that the design and framework followed for software development are appropriate to support its reuse.

b) Testable

Software is testable, when it supports acceptable criteria and evaluation of performance. The reuse must be able to be tested i.e. we must be able to generate reusable test cases for the reuse process. Thus, to reuse software cost-effectively, we must re-verify components in their new environment.

c) Stable

It should be a stable process and it should remain for definite period. One important question to consider is: Are my software development environments, tools and platforms well defined and stable? If not, I should first focus on domain models and business requirements.

Scenarios:

Scenario #: Client owns some deposit in the bank.

The deposit is kind of asset, which is saved in the bank. In this scenario, the clients save the deposit and the bank manages it. In addition, the bank has some specific criteria to describe the deposit. When the client wants to proof that they own the deposit, the bank will provide some evidence to show the state and other related information about the deposit. Bank reuses the collateral pledged by its clients, as collateral for its own borrowing.

Solution

DRAFT

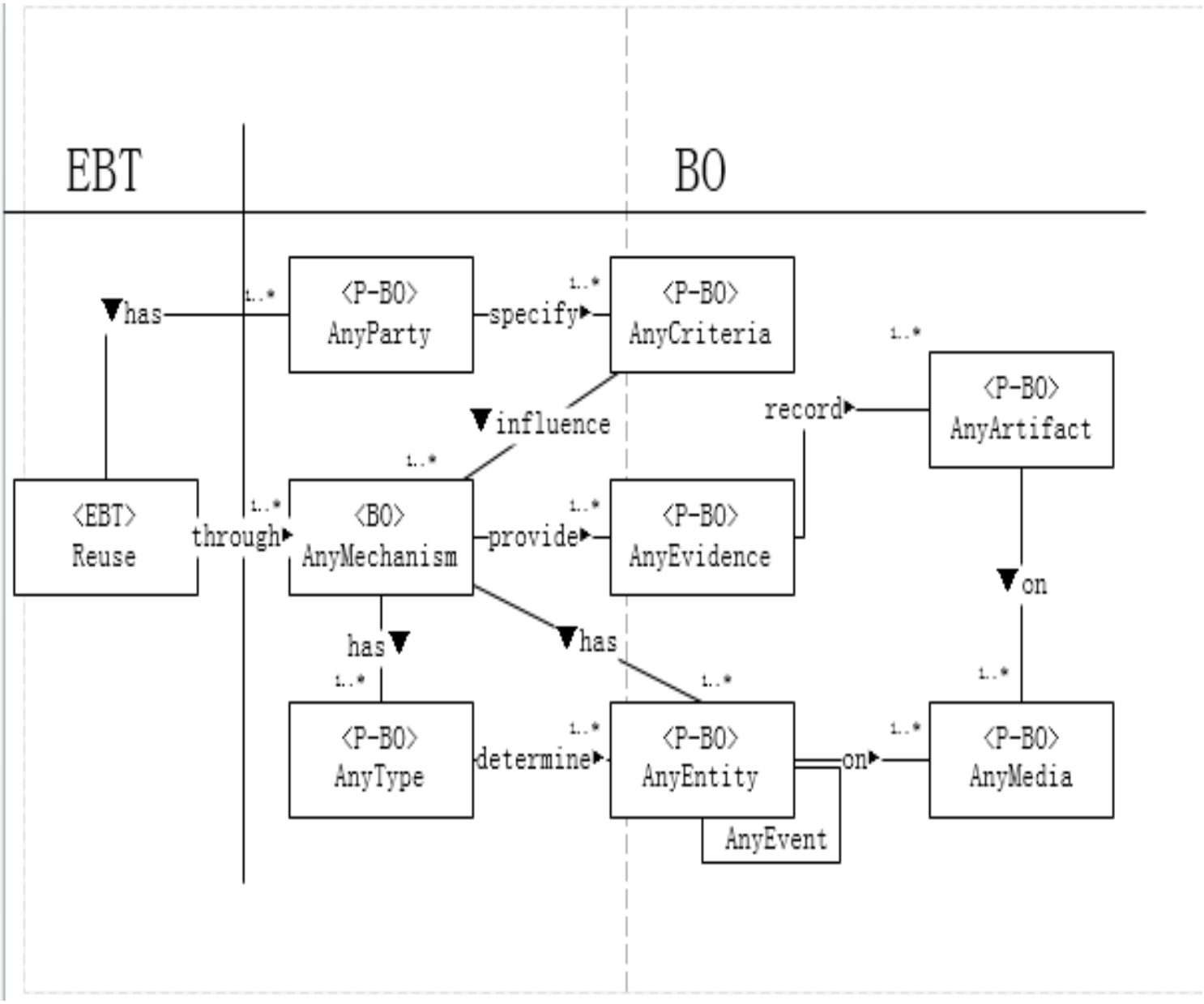


Figure 4.2: Reuse Stable Analysis Pattern

Description:

DRAFT

The Party requests a reusability of an Asset like code to produce an Outcome. In this process, the Party needs to use some Mechanism to produce the Outcome. Moreover, the Context determines the Entity on the Medium.

4.3 Application

Application #1: Client owns some deposit in the bank and bank reuses the collateral pledged by its clients as collateral for its own borrowing.

2)

Table 2.5: Asset Applications #1

EBT	BOs	APP-1
Reuse	----------	----------
	AnyAsset	Deposit
	AnyParty	Client, Bank
	AnyCriteria	SpecificCriteria
	AnyType	SpecificType
	AnyEvidence	SystemLog
	AnyEntity	SpecificDeposite
	AnyArtifact	Certification
	AnyMedia	BankSystem

Case Studies
Scenario #1
Description:
The client saves the deposit. The deposit has the type and the type determines the entity. The bank follows the criteria to record the log on the bank system and produce the certification.

Model:

1) The **AnyParty (Client)** has the **Ownership** of the **AnyAsset (Deposit)**.
The **AnyAsset (Deposit)** has **AnyType (SpecificType)**.
The **AnyType (SpecificType)** determines the **AnyEntity (SpecificDeposit)**.
The **AnyEntity (SpecificDeposit)** is stored by **AnyParty (Bank)**.
The **AnyParty (Bank)** follows the **AnyCriteria (SpecificCriteria)** to record **AnyLog (SystemLog)** on **AnyMedia (BankSystem)**.
The **AnyMedia (BankSystem)** returns the **AnyEvidence (Certification)** to the **AnyParty (Client)**.

4.4 Class Diagram:

DRAFT

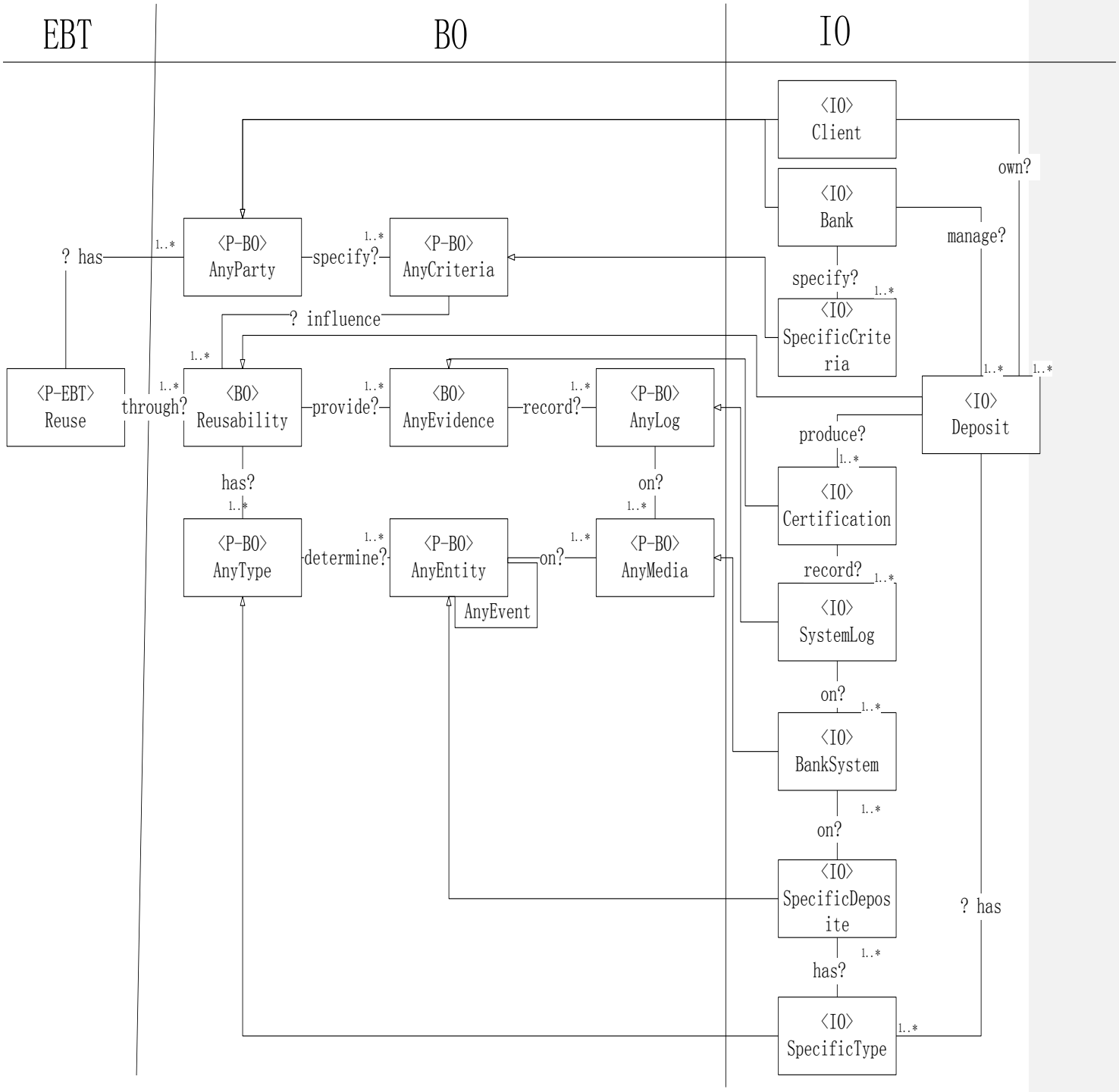


Figure 4.3: Reuse Application #1

Use Case:

DRAFT
UC #:1
UC title: Saving the deposit
Actor: Person (Client), Organization (Bank)
UC Description:
The client saves the deposit in the bank.
The client mortgages the house in the bank.

UC #: 2
UC title: Recording log
Actor: Organization (Bank), Software (BankSystem)
UC Description:
The bank follows the criteria to evaluate and produce some log in the bank system.
Sequence diagram:

4.5 Sequence Diagram:

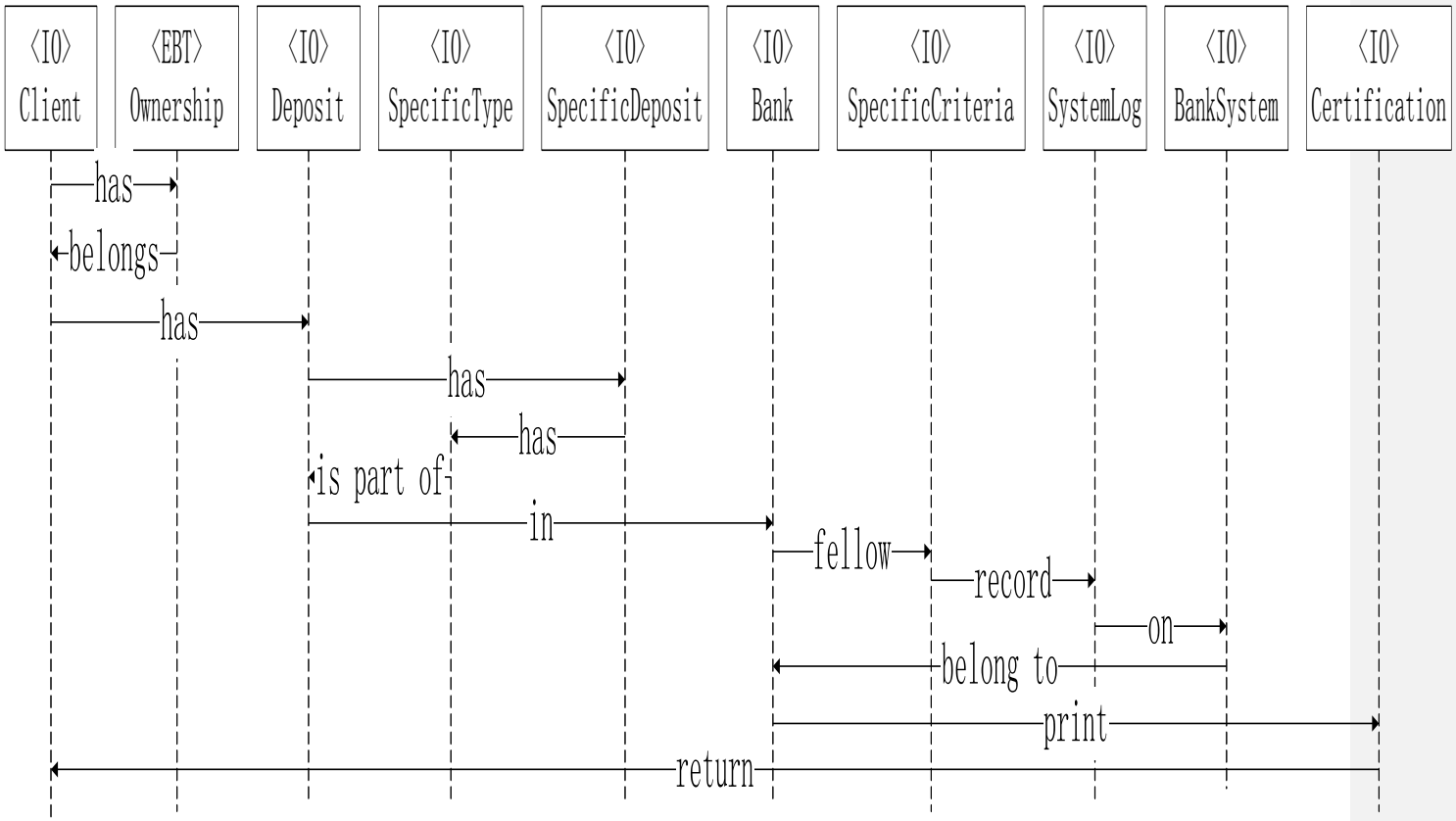


Figure 4.4: Sequence Diagram

V) SUMMARY

DRAFT

Anyone who has spent time building software in an organization will tell that achieving software reuse is extremely challenging. Large scale, systematic reuse is even harder in an organization. A developer, with tight deadlines to meet and complete functionality to deliver, may find it challenging to keep reuse as a top priority.

Going through the process of creating this knowledge map, we have come to realize that software reuse is a viable idea. That is not to say, it is easy and there are several ways to spoil software reuse. Lack of leadership and vision for making the effort work within the organization's political and cultural context and non-alignment with what business sponsors are trying to accomplish is a key factor. Some efforts might fail, because they are overly ambitious, where many big upfront design efforts are spent trying to design things future perfect. Still others fail due to lack of design flexibility, inadequate planning, and funding issues. Communication effectiveness and awareness of existing reusable software assets is also a critical factor.

The purpose of this Paper is to present a few tips on succeeding with systematic reuse based on the given knowledge map. Following have been our key learning points while carrying out this work:

#1 - Focus on domain-specific software assets
Business assets are what makes an application or product line unique, the organization special, and ultimately differentiate it from the existing competition. The faster one can develop, release, and iteratively improve domain relevant software assets the faster one will meet changing business needs and serve customers with complete satisfaction.

#2 Name software assets appropriately
Whether one is trying to name a method, a class, a component, a library, or a service, pause a brief minute to think about the software's purpose and capabilities in order to assign a name. An appropriate name will help when trying to mine for existing software assets to be reused. Additionally, the effort will be fruitful when one is trying to re-factor existing software assets to be more reusable.

#3 Evolve Reusable Assets Iteratively
When one is recognizing the need for a reusable software asset, it is a key to map out a realization strategy. If one approaches asset realization in a big way, he/she could end up creating a software asset that is irrelevant to the project's immediate needs and it will add significant schedule risk due to increased design, development, and testing time. Either way, a developer may spend plenty of precious resources.

DRAFT

VI) REFERENCES

[1] https://en.wikipedia.org/wiki/Code_reuse

[2] ML Griss, Systematic Software Reuse - Objects and frameworks are not enough, Object Magazine, February 1995.

[3] Ivar Jacobson, Martin Griss and Patrik Jonsson, Software Reuse: Architecture, Process, and Organization for Business Success, Addison-Wesley 1997 (to be published).

[4] J Hooper & R Chester, Software Reuse Guidelines and Methods, Plenum, 1991.

[5] EA Karlson, Software Reuse: A holistic approach, Wiley 1995.

[6] W Schäfer, R Prieto-díaz and M Matsumoto, Software Reusability, Ellis Horwood, 1994.

[7] ML Griss and K Wentzel, Hybrid Domain Specific Kits, Journal of Systems and Software, Dec 1995.

[8] ML Griss and RR Kessler, Building Object Oriented Instrument Kits, Object Magazine, May 1996.

[9] R Malan and T Dicolen, Risk Management in an HP Reuse Project, Fusion Newsletter, April 1996

[10] W Frakes and S Isoda, Systematic Reuse, Special Issue IEEE Software, May 1994.

[11]] M.E. Fayad, Mohamed, Sanchez, Huascar , Hegde, Srikanth, Basia, Anshu, and Vakil, Ashka, Software Patterns, Knowledge Maps, and Domain Analysis. Auerbach Publications (December, 2014), 422 pages.

[12] Fayad, Mohamed and Altman, Adam. "An Introduction to Software Stability." The Communications of the ACM, Vo. 44, No. 9, September 2001, pp 95-98.

[13] Fayad, Mohamed. "Accomplishing Software Stability.*"The* Communications of the ACM, Vo. 45, No. 1, January 2002, pp 95-98.

[14] Fayad, Mohamed "How to Deal with Software Stability." The *Communications of ACM* vol. 45, no. 4, pp. 109-112, Apr. 2002.

[152] Fayad, M.E. Stable Analysis Patterns: Understanding of the Problem, vrlSoft publishing, December 2916.

DRAFT

[16] Fayad, M.E. Stable Design Patterns: The Ultimate Solutions, vrlSoft publishing, December 2916.

IX) APPENDIX 1.: KNOWLEDGE MAP TEMPLATE

9.1 KM Name: Software Reuse

9.2 KM Nickname: None

9.3 KM Domain/Subject/Topic Description : Software reuse or/Topic ATEhimede/fichiers/23356/ch04software, or software knowledge, to build new software [1] by byr software knowledge, to build new sof**.** 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 [3].

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 [4]. 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 even between products in different product lines. The scope of reuse is limited, in general, because of the nature of and constraints on a product line; for example, it is unwise to reuse a desktop application in a mission-critical system.

9.4 (EBTs/Goals):

Table 1.1: EBTs of "SOFTWARE REUSE"

Sr. no.	EBTs/Goals	Description
1.	Reuse	To reuse something is to use it again, after it has been used once. This includes conventional reuse, where the item is used again for the same function and creative reuse, where it is used for a different function.
2.	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.
3.	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.
4.	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.
5.	Stability	Stable software is so named because it is unchanging. Its behavior, functionality, specification or API is considered t to execute only one aspe 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.
6.	Stop Reinventing the wheel	By reusing software, we make sure that we don't waste time and resources in reinventing what is already done and available to us. This paves way for more investment in innovation.
7.	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.
8.	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.
9.	Development	Software development is the process of programming, documenting, testing and bug-fixing various applications and frameworks in order to build a software product.
10.	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.
11.	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.

12.	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.

9.5 BOs/Properties:

Table 1.2: BOs of "**SOFTWARE REUSE**"

Sr. no.	BOs/Capabilities	Description
1.	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.
2.	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 that we are using bug-free component to build new software.
3.	Outcome	A result or a consequence resulting from an action. The final software that includes the reusable parts is highly dependable software that adheres to the standard compliances.
4.	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.
5.	Medium	An intervening substance or agency for transmitting or producing an effect. The platform on which the existing resource and final software plays on.

DRAFT

6.	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 software for the parties separately.
7.	Asset	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.
8.	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, we have to follow a systematic approach towards it.
9.	Frequency	Frequency is the state of being frequent or occurring often. Frequency with which we are using software is an important metric for defining its reusability.
10.	Level	One of the promises of object-orientation is reuse. Developing new software systems is expensive, and maintaining them is even more expensive. Reuse is therefore sensible in both business and technology perspectives. We must carefully identify the levels in software reuse.
11.	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.
12.	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.

DRAFT

13.	Use Cases	We can 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.
14	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.
15	Methods	Well-defined methods are of utmost importance to achieve reusability in software. These are the principles or guidelines that give a direction to the entire process of software reuse. Thus, methods play a vital role.
16	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 describe the software system. These are further used, when we reuse software.
17	Polymorphism	It is the ability of having, or assuming, various forms, characters, or styles. One of the major goals of OOP is software reuse. We can illustrate this by considering two different approaches to reuse: Inheritance -- *is-a* relationship. Composition -- *has-a* relationship.
18	Constraint	A limit or restriction of using the existing resource in the design. There can be various factors restricting the reuse of software that can range from lack of tool support to involved maintenance costs.
19.	Contract	Software component is associated with a contract that 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.
20.	Defect	Defect is a shortcoming or imperfection in software that puts limitations on its working or functionalities. Software reuse improves quality of software leading to reduction in defect density in it.

DRAFT

21.	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 leads to reduced process risks.
22.	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.
23.	Resource	Resource defines the materials, strategies etc. undertaken for the completion of a task. Software reuse leads to the saving of resources largely. Reusable resources can be template, component, framework, artifact, pattern apart from the reuse of code and inheritance.
24.	Project	Project outlines a detailed plan to accomplish a task involving considerable goals, resources, cost and personnel. Reuse, between projects is where, we can think of taking the greatest advantage.
25.	Function	A basic task of a computer, especially one that corresponds to instructions from the user. The function that the party needs and the existing resource can implement can vary from one party to another and accordingly, the functionalities can be implemented in software for the parties separately.
26.	Scenario	A scenario is an outline of a subsystem. It includes a sequence of possible events to be studied in a system of interest.
27.	Model	An abstract system, obeying certain specified conditions, which purpose is to study or illustrate an entity or event.
28.	Library	A collection of standard programs and subroutines for immediate use
29.	Component	One element of a large system, which purpose is to implement a specific function.
30.	Object	A self-contained module of data and its associated processing. Objects are the software building blocks of object technology.

DRAFT

31.	Diagram	A schematic representation of a sequence of subroutines designed to solve a problem
32.	Class	A description of the structure and operations of an object. Any one of this collection share the same characteristics.
33.	Layer	Architects should look to reuse significant application frameworks, such as layers that can be applied to many different types of applications.

9.6 Knowledge Map (Core Knowledge):

Table 1.3: Knowledge Map of "Software reuse"

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	AnyParty, AnyMechanism, AnyType, AnyEntity, AnyEvent, AnyCriteria, AnyArtifact, AnyMedia, AnyEvidence
Abstraction	AnyParty, AnyType, AnyCriteria, AnyEvidence, AnyEntity, AnyEvent, AnyMedia, AnyLog
Ownership	AnyParty, AnyActor, AnyMechanism, AnyContext, AnyCriteria, AnyLog, AnyType, AnyEntity, AnyEvent, AnyMedia
Encapsulation	AnyParty, AnyType, AnyScenerio, AnyCriteria,

DRAFT

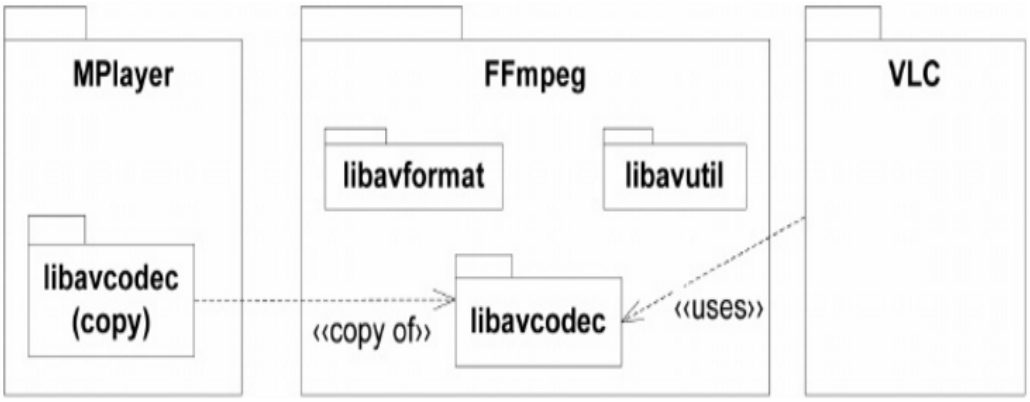
	AnyEntity, AnyEvent, AnyApplication, AnyMedia

X) APPENDIX 2. CASE STUDIES OF SOFTWARE REUSE

Some case studies on Software Reuse

10.1 Case Study: Software Reuse in open Source

One of the most significant ways that commercial developers of software can become more efficient and productive is to reuse not only software, but also best practices from the open-source movement. The open-source movement encompasses a wide collection of ideas, knowledge, techniques, and solutions. Commercial software vendors have an opportunity to both



learn from the open-source community, as well as leverage that knowledge for the benefit of its commercial clients.

Figure 9.1: Black-box reuse (by VLC) and white-box reuse (by Mplayer)

As an example shown in the above diagram, the Mplayer project keeps a copy of the library in its repository (and eventually modifies it for its own purpose as in white-box testing), while in the VLC Project, at compilation time needs user to provide the location of up-to-date version of Ffmpeg project (black-box reuse).

This knowledge map looks at a number of the characteristics of the open-source movement by including software reuse and abstraction. Code reuse is a form of knowledge reuse in software development, which is fundamental to innovation in many fields. Yet, to date, there has been no systematic investigation of code reuse in open source software projects. Thorugh our study, we find that code reuse is extensive across the sample. And, that open source software developers, much like developers in firms, apply tools that lower their search costs for knowledge and code and assess the quality of software components, they have incentives to reuse code. Open source

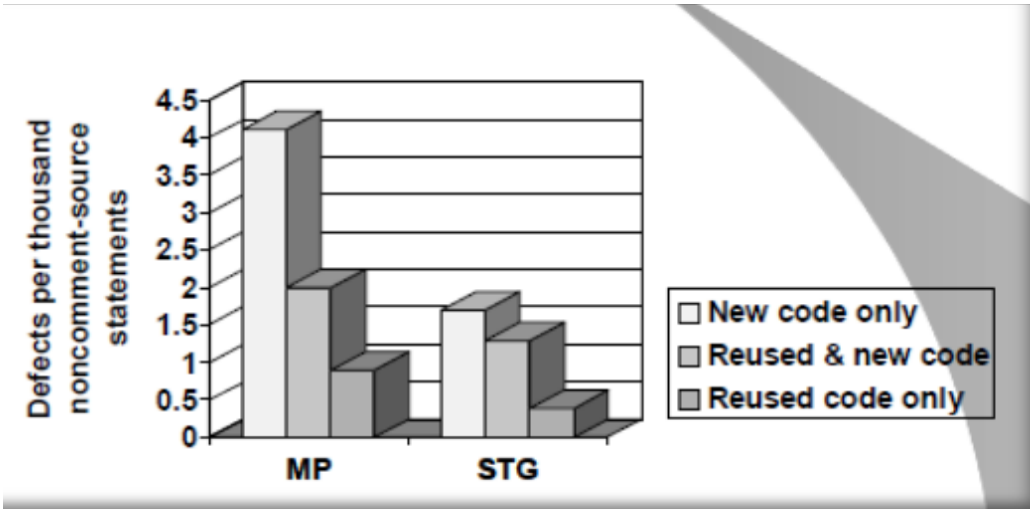
DRAFT

software developers reuse code, because they want to integrate functionality quickly, write preferred code, operate under limited resources in terms of time and skills, and mitigate development costs through code reuse.

10.2 Case study: Software Reuse in Hewlette Parckard

To prove this concept true that reuse increase quality and productivity, case studies were performed at HewletteParkard. Hewlette- Parckard found that reuse significantly play a positive effect on the software development. The case studies presented two metrices from two HP reuse programs that showed that with the quality improvement the productivity is also increased. The first case study was carried out in the Manufacturing Productivity section of HP's software technology division. The MP section produces large-application software for manufacturing resource planning. The study was started in 1983. Originally, the motive of this was just to increase the engineering productivity. But the MP section has discovered reuse for maintaining the burden and support product enhancement.

- MP engineers practiced reuse by using generated code and other work products such as applications and architecture utilities and files.
- The data reported only the use of reusable workproducts, not the generated code.
- Total code size for the 685 reusable work products was 55,000 lines of non-comment source



statements.

Figure 10.1: Code quality graph