Abstract - this paper surveys the different approaches to software reuse found in the research literature. It describes and compares the different approaches and makes generalizations about the field of software reuse.

In this survey we will present the definitions of software reuse and will demonstrate the cases where software reuse are valuable. Then, the different approaches of software reuse are mentioned with an examination of the effectiveness of each approach. Subsequently, the advantages and disadvantages of each approach are presented. After that, we will study the difficulty of implementing a software reuse process. Finally, the open areas of research in this field are highlighted.

Keywords: Software Reuse, Architecture, COTS, Design Patterns, Requirement, and Product Line

1 Introduction

This paper will make a breadth survey about the different approaches of software reusability.

Let us start with the basic definitions. We have two different terms related with Software Reusability Development.

1.1 Software Development with Reuse: Software development with reuse is the use of existing software or software knowledge to construct new software. Reusable assets can be either reusable software or software knowledge. In this survey we will focus on this part of reuse the software development with reuse [1].

1.2 Software Development for Reuse: Software Development for Reuse is a process of producing potentially reusable components. We know clearly the difficulties that are faced when trying to reuse a component that is not designed for reuse. The process of developing potentially reusable components depends solely on defining their characteristics such as language features and domain abstractions [2].

However, both terms are overlapped related to the whole Reuse process.

Reusability is a property of a software asset that indicates its probability of reuse. Software reuse’s purpose is to improve software quality and productivity. Reusability is one of the major software quality factors. Software reuse is of interest because people want to build systems that are bigger and more complex, more reliable, less expensive and that are delivered on time [1].

1.3 Software Reuse Benefits [3]:

• Increased dependability: Reused software, that has been tried and tested in working systems, should be more dependable than new software.

• Reduces Process Risks: If software exists, there is less uncertainty in the costs of reusing that software than in the costs of development. This is an important factor for project management as it reduces the margin of error in project cost estimation.

• Effective use of specialists: Instead of application specialists doing the use of specialists same work on different projects, these specialists can develop reusable software that encapsulate their knowledge.

• Standards compliance: Some standards, such as user interface standards, can be implemented as a set of standard reusable components. For example, if menus in a user interfaces are implemented using reusable components, all applications present the same menu formats to users.

• Accelerated development: Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time should be reduced.

1.4 Software Reuse Problems [4]:

• Increased maintenance costs: If the source code of a reused software system or component is not available then maintenance costs may be increased as the reused elements of the system may become increasingly incompatible with system changes. Lack of tool support CASE toolsets may not support development with reuse.

• Not-invented-here syndrome: Some software engineers sometimes prefer to re-write components as they believe that
they can improve on the reusable component. This is partly to do with trust and partly to do 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 can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.

• 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 make routinely include a component search as part of their normal development process.

1.4 Software Reuse Activity [4]: The reuse activity is divided into six major steps performed at each phase in preparation for the next phase. These steps are:

1. Studying the problem and available solutions to the problem and developing a reuse plan or strategy;
2. Identifying a solution structure for the problem following the reuse plan;
3. Reconfiguring the solution structure to improve reuse at the next phase;
4. Acquiring, instantiating, and/or modifying existing reusable components;
5. Integrating the reused and any newly developed components into the products for the phase, and

2. Software Product Line (SPL):

The study of software product lines addresses the issues of engineering software system families, or collections of similar software systems. The objective of a software product line is to reduce the overall engineering effort required to produce a collection of similar systems by capitalizing on the commonality among the systems and by formally managing the variation among the systems. This is a classic software reuse problem [5].

2.1 Basic Software Product Line Concepts [6]:

Software product lines can be described in terms of four simple concepts, as illustrated in the figure below:

Software asset inputs: a collection of software assets – such as requirements, source code components, test cases, architecture, and documentation – that can be configured and composed in different ways to create all of the products in a product line. Each of the assets has a well-defined role within a common architecture for the product line. To accommodate variation among the products, some of the assets may be optional and some of the assets may have internal variation points that can be configured in different ways to provide different behavior.

Decision model and product decisions: The decision model describes optional and variable features for the products in the product line. Each product in the product line is uniquely defined by its product decisions - choices for each of the optional and variable features in the decision model.

Production mechanism and process: the means for composing and configuring products from the software asset inputs. Product decisions are used during production to determine which software asset inputs to use and how to configure the variation points within those assets.

Software product outputs: the collection of all products that can be produced for the product line. The scope of the product line is determined by the set of software product outputs that can be produced from the software assets and decision model.

![Figure 1: Basic Software Product Line Concepts](image)

2.2 Software Product Line Challenges:

However, the predominant challenges, in most software product lines, are:

a) The management of variability required to facilitate the product differences. This is due to the fact that industrial software product lines can easily incorporate thousands of variable features and configuration parameters for product customization. Managing this amount of variability is extremely complex. One of the reasons for this high complexity is that, due to continuous evolution of the product line, a large number of new variable features and configuration parameters are introduced but at the same time obsolete variability is not removed. This increasing complexity results in a combinatorial explosion of variants [7].

b) With single systems, software engineers can maintain a single point of view throughout the development process (i.e., focused on the implementation of the single system). In contrast, with software product lines, software engineers must take different points of view at different times in order to effectively develop the software family[5].
However, there are several tools exist for support Software Product Line development and maintaining such as ConExp, sunifdef and DMS [5].

3 Commercial of the Shelf (COTS):

A commercial-off-the-shelf (COTS) product is a software system that can be adapted to the needs of different customers without changing the source code of the system [3].

When a software system is developed around a COTS product[8], it is called a “COTS-solution system.” If a system includes a large proportion of COTS products it is called "COTS- intensive systems"], "COTS-integrated systems"[2], or "COTS-aggregate systems" [9]. However, the term “a COTS-based system” is generally used for all purposes [8].

3.1 COTS-Solution System:

A COTS-solution system is a single product or suite of products, usually from a single vendor, that can be tailored to provide the system’s functionality. Vendors offer such solutions if a consistent and well-bounded range of end-user needs exists throughout a broad community, justifying the vendors’ costs for developing the products or suites of products.

Significant tailoring is required to set up and use these products, and the ability and willingness of an organization to understand and adopt the processes supported by the products are often key factors in success or failure. COTS-solution systems are commonly found in such well-established domains as personnel management, financial management, manufacturing, payroll, and human resources. Typical software vendors in this area include PeopleSoft, Oracle, and SAP.

COTS-Solution Systems usually require extensive configuration to adapt them to the requirements of each organization where they are installed. Once the configuration settings are completed, a COTS-solution system is then ready for testing. Testing is a major problem when systems are configured rather than programmed using a conventional language [9].

3.2 COTS-Integrated Systems:

COTS-aggregate systems are systems in which many disparate products (from different and sometimes competing vendors) are integrated to provide a system’s functionality. Such systems are created if operational procedures are sufficiently unique to preclude the possibility of a single.

COTS product solution, if the constituent technologies are immature, if the scale of the system is large enough to encompass several domains, or simply because different products provide distinct pieces of functionality to form the complete system. Systems with these characteristics include software support environments, large information systems, and command-and-control systems. Often, the COTS products and other components are combined in ways or to degrees that are unprecedented [3].

3.2.1 Challenges of COTS-Integrated System:

While adapting these components we did not care to identify whether the causes of our problems were with the functionality of the COTS products, their architecture, or in fact the functionality or architecture we desired, so it is somewhat difficult to button-hole the problems easily. [10].

3.3 Main Processes for Evaluation and Selecting COTS Software [11]:

Based on previous studies, several processes for evaluating and selecting COTS software are shared by existing methods for COTS software selecting. These processes can be ordered as iteratively, sequentially, or overlapping. However, the common processes for evaluating and selecting COTS software can be classified in terms of four general processes.

Supporting Process : This process consists of set of activities that support other processes of the valuation and selection. This process begins with planning for an evaluation and selection COTS software; the tasks that might be completed during this activity include forming the evaluation and selection team (e.g. technical experts, domain experts, end users, etc.), identifying stakeholders (e.g. integrators, (funding customers, business owners, etc.), define the goals and objectives, etc. Documentation is also performed during this process.

Preparation Process: The main purpose of this process is to collect and prepare the information that required for further detail evaluation.

Evaluation Process: This process plays a vital role to determine how well each of the COTS software alternatives achieves the evaluation criteria.

Selecting Process: The outputs of the evaluation process are several kinds of data such as facts, checklists, weights, opinions. Those kinds of data should be consolidated and interpreted into information.

3.4 COTS Products Problems [8]

Incompatibility: COTS component may not have the exact functionality required; moreover, a COTS product may not be compatible with in-house software or other COTS products; Inflexibility: usually the source code of COTS software is not provided, so it cannot be modified; Complexity: COTS products can be too complex to learn and
to use imposing significant additional effort; Transience: Different versions of the same COTS product may not be compatible, causing more problems for developers.

4. Software Requirement Reuse

Much of the effort of building complex software systems goes into understanding, specifying, and validating system requirements. For mission- and safety critical systems, requirements errors represent a major source of development problems. Prior work in product-line engineering has shown that we can substantially increase productivity while decreasing errors by systematically re-using (rather than recreating) the work products for families of systems where system requirements are sufficiently similar. Embedded software for commercial product lines like printers, mobile phones, or flight-control systems are typically families in this sense [12].

4.1 Requirement Representation for Reuse

The systematic requirements reuse to develop software requires two specific actions. First, to define the adequate way to model and to store specifications in the phase of development for reuse. Second, to define a process to compare and to adapt the reusable requirements in the phase of software development with reuse [13].

4.2 Comparing and Adapting Requirement

Comparing and adapting requirements means that it should be established an equivalence relation between requirements models and the sufficient condition to determine the similarity between the requirements models, and it should be established a process to compare requirements so that it supports software. One technique is to reuse domain descriptions and task specifications. And the other is to apply techniques based on artificial intelligence to support the structural and semantic matching when retrieving requirements [13].

4.1 Benefits of Requirement Reuse

Requirements need not be re-Validated with stakeholders repeatedly; ensure consistency of requirements & business rules within organization or Program; test Cases/Test coverage is already available and can be reused; reduce requirements work for subsequent uses.

However, the main obstacle reported for adopting requirements reuse is poor quality of existing requirements. Having unstructured, incomplete, outdate existing requirements makes it difficult to reuse them going forward. Developing techniques to analyze the inventory of and refactor existing requirements can help practitioners better adopt and benefit from reuse.

5. Code Reuse

In computer science and software engineering, reusability is the likelihood a segment of source code that can be used again to add new functionalities with slight or no modification. Reusable modules and classes reduce implementation time, increase the likelihood that prior testing and use has eliminated bugs and localizes code modifications when a change in implementation is required.

The evolution of programming languages is tightly coupled with reuse in two important ways. First, programming languages have evolved to allow developers to use ever larger grained programming constructs, from ones and zeroes to assembly statements, subroutines, modules, classes, frameworks, etc. Second, programming languages have evolved to be closer to human language, more domain focused, and therefore easier to use. Languages such as Visual C++, Delphi, and Visual Basic clearly show the influence of software reuse research [1].

To reduce programming effort and shorten time-to-market, programmers can find and reuse existing solutions for their prototypes. Source code search engines have been developed to locate implementations that are highly-relevant to a feature specified by a programmer (e.g., via a natural-language query). Existing search engines often return packages that match only a small subset of the desired
features, and developers have to invest considerable effort to integrate features from several different packages and projects. Under these circumstances, the cost and effort required for a programmer to comprehend and integrate the returned source code can significantly reduce the benefits of reuse [14].

But, code, which is executed from other developer, has a problem hard to understand and reuse because of missing and insufficient document, the existing system developer's absence. And that code causes decline in performance. It also needs much time and costs in order to solve these problems [15].

5.1 Code Reuse Benefits [16]

Reusing code saves programming time, which reduces costs. Sharing code can help prevent bugs by reducing the amount of total code that needs to be written to perform a set of tasks. Relatedly, separating code into specialized libraries lets each be tuned for performance, security, and special cases. Delegation of tasks into shared modules allows offloading of some functionality onto separate systems. Proper and efficient reuse of code can help avoid code bloat. Bloated code contains unnecessary duplication and unused instructions.

5.1 Code Reuse Drawbacks [16]

There are other potential drawbacks to code reuse, often very dependent on the situation and implementation:

5.1.1 Performance might become a factor:

Depending on the platform and programming language, a library or framework might perform slower than desired. In some situations it might be beneficial to build a specialized one-time solution instead of using a common library. APIs accessed over a network will sometimes be slower than solving a problem within the local system. The system of modularity itself might create a bottleneck. For example, extra process initialization or shared library management can create overhead.

5.1.2 Loss of control over 3rd party solutions might have negative repercussions.

For example, there might be lack of support, desired feature enhancements might not get added, or security might not be fully tested. Outside the technical considerations, there might also be licensing and liability issues. When not well implemented or when taken too far, code reuse can eventually cause code bloat. Ironically, adding modularity can eventually lead to lingering APIs and libraries which go unused. In very large systems it's not uncommon to lose track of how every component is used. Over time a component my become useless, but linger in the system. This, however, is not so much an inherent drawback of code reuse as it's a problem of implementation.

6. Design Reuse

Broadly speaking, design reuse appears promising for at least three reasons. First, since designs address early phases of system development, many of the up-front (and hence most costly) errors can be avoided. Second, reuse of familiar designs can improve the understandability of a system, making it easier to evolve and maintain. Third, design reuse promotes code reuse: often much of the infrastructure to support a design can be shared among applications that share that design.

It is perhaps not surprising then, that some of the more impressive examples of reuse today involve a strong component of design reuse. Prominent examples include specialized frameworks such as user interface toolkits, application generators (such as Visual Basic), domain specific software architectures, and object-oriented patterns [17].

6.1 Framework Reuse

A software framework is an abstraction in which software providing generic functionality can be selectively changed by user code [clarify], thus providing application specific software. A software framework is a universal, reusable software platform used to develop applications, products and solutions. Software frameworks include support programs, compilers, code libraries, an application programming interface (API) and tool sets that bring together all the different components to enable development of a project or solution [18].

6.1.1 Framework Reuse Benefits [18]

Application frameworks offer a variety of advantages:

Using code which has already been built, tested, and used by other programmers increases reliability and reduces programming time. Software development teams can be split between those who program the framework and those who program the final complete application. This separation of tasks lets each team focus on more specific goals and use their individual strengths. Frameworks can provide security features which are often required for a common class of applications. This provides every application written with the framework to benefit from the added security without the extra time and cost of developing it. By handling “lower level” tasks frameworks can assist with code modularity. Frameworks often help enforce platform-specific best practices and rules. Frameworks can assist in programming to design patterns and general best practices. Upgrades to a framework can enhance application functionality without extra programming by the final application developer.
6.1.1 Framework Reuse Drawbacks [18]

There can be negative consequences to using a framework:

Performance can sometimes degrade when common code is used. This sometimes occurs when a framework must check for the various scenarios in which it is used to determine a path of action. Frameworks often require a significant education to use efficiently and correctly (i.e. some have a high learning curve). Functionality which needs to bypass or work around deficiencies in a framework can cause more programming issues than developing the full functionality in the first place. Bugs and security issues in a framework can affect every application using that framework. Therefore it must be tested and patched separately or in addition to the final software product.

6.2 Architecture Reuse [17]

The other broad area of related work is software design reuse, a topic that is receiving increasing attention from researchers and practitioners in areas such as module interface languages, domain-specific architectures, software reuse, codification of organizational patterns for software, architectural description languages, formal underpinnings for architectural design, and architectural design environments. Collectively these efforts are attempting to establish an engineering basis for architectural design, and make principles and techniques of architectural design more widely accessible.

6.2.1 Software Architecture and Architecture Styles [17]

An architectural style provides a specialized architectural design vocabulary for a family of systems, and typically incorporates a number of idiomatic uses of that vocabulary and design rules for system composition. From the point of view of a designer, architectural style is important for several reasons:

It limits the design space, thereby simplifying design choices. It allows a designer to exploit recurring patterns of organization, such as topological configurations, or even specific organizations of components (such as the MVC pattern in object-oriented systems). It provides a context within which certain kinds of design integrity can be enforced, such as the fact that no cycles are allowed. It permits specialized analyses such as detection of deadlock. And finally, as we detail in the next section, it provides a basis for supporting reuse of architectural building blocks and patterns.

6.3 Design Patterns

A design pattern 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. Patterns are formalized best practices that the programmer must implement themselves in the application. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Many patterns imply object-orientation or more generally mutable state, and so may not be as applicable in functional programming languages, in which data is immutable or treated as such.

6.3.1 Design Patterns Goals:

- To support reuse of successful designs, to facilitate software evolution (add new features easily, without breaking existing ones), in short, we want to design for change.

6.3.2 Types of Design Patterns

- **Creational Patterns**: To create objects rather than developer instantiate it.

- **Structural Patterns**: to compose group of objects in larger structures.

- **Behavioral Patterns**: To defines communication & flow between objects.

7. Conclusions

From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language. The only significant reuse or software was the reuse of functions and objects in programming language libraries. However, costs and schedule pressure meant that this approach became increasingly unviable, especially for commercial and Internet-based systems. Software reuse is possible at a number of different levels:

1. **The abstraction level**: At this level, you don’t reuse software directly but rather use knowledge of successful abstractions in the design of your software. Design patterns and architectural patterns are ways of representing abstract knowledge for reuse.

2. **The object level**: At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need. For example, if you need to process mail messages in a Java program, you may use objects and methods from a JavaMail library.

3. **The component level**: Components are collections of objects and object classes that operate together to provide related functions and services. You often have to adapt and extend the component by adding some code of your own. An example of component-level reuse is where you build your
user interface using a framework. This is a set of general object classes that implement event handling, display management, etc. You add connections to the data to be displayed and write code to define specific display details such as screen layout and colors.

4. The system level: At this level, you reuse entire application systems. This usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system’s own configuration interface. Most commercial systems are now built in this way where generic COTS (commercial off-the-shelf) systems are adapted and reused. Sometimes this approach may involve reusing several different systems and integrating these to create a new system. By reusing existing software, you can develop new systems more quickly, with fewer development risks and also lower costs. As the reused software has been tested in other applications, it should be more reliable than new software.

However, there are costs associated with reuse:

1. The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. You may have to test the software to make sure that it will work in your environment, especially if this is different from its development environment. 2. Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high. 3. The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing. 4. The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed. Integrating reusable software from different providers can be difficult and expensive because the providers may make conflicting assumptions about how their respective software will be reused.

How to reuse existing knowledge and software should be the first thing you should think about when starting a software development project. You should consider the possibilities of reuse before designing the software in detail, as you may wish to adapt your design to reuse existing software assets.

8. References


[19] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides " Design Patterns: Elements of Reusable Object-Oriented Software techniques".