International Journal of Computer Science and Informatics

Volume 1 | Issue 2 Article 3

October 2011

Software Reusable Components With Repository System

Chintakindi Srinivas
Department of CSE Kakatiya Institute of Technology and Sciences Warangal, India,
chintakindisrinivas77@gmail.com

Recommended Citation
Srinivas, Chintakindi (2011) "Software Reusable Components With Repository System," *International Journal of Computer Science and Informatics*: Vol. 1 : Iss. 2 , Article 3.
DOI: 10.47893/IJCSI.2011.1016
Available at: https://www.interscience.in/ijcsi/vol1/iss2/3

Software Reusable Components With Repository System

Chintakindi Srinivas
Associate Professor, Department of CSE
Kakatiya Institute of Technology and Sciences
Warangal, India
E-mail: chintakindisrinivas77@gmail.com

Dr.C.V.Guru rao
Principal and Professor of CSE
Kakatiya Institute of Technology and Sciences
Warangal, India

Abstract

Software reuse is the process of creating software systems from existing software rather than building them from scratch. The goal is the use of reusable components as building blocks in new systems with modifications occurring in a controlled way. The reuse of software components is the key to improve productivity and quality levels in software engineering. One of the most promising approaches to reduce costs and improve reliability is component-based development, which aims to allow new applications to be assembled from prefabricated parts rather than coded from scratch. Software reuse involves building software that is reusable by design and building with reusable software. Software reuse includes reusing both the products of previous software projects and the processes deployed to produce them, leading to a wide spectrum of reuse approaches, from the building blocks (reusing products) approach, on one hand to the generative or reusable processor (reusing processes), on the other.

Keywords- Software Reuse, Component Based Software Engineering.

1. Introduction

A Software reuse enables developers to leverage past accomplishments and facilitates significant improvements in software productivity and quality. There are several motivations for desiring software reuse, including gains in productivity by avoiding redevelopment and gains in quality by incorporating components whose reliability has already been established. Reuse-based software development emphasizes strategies, techniques, and principles that enable developers to create new systems effectively using previously developed architectures and components.

Software reuse offers a great deal of potential in terms of software productivity and software quality by dealing with software products at the component level. Also by focusing on abstract descriptions of software components, it addresses the question of scale; on the other hand, by dealing with software design at the architectural level, rather than the coding level, it addresses the question of emphasis.

Developing with reusable assets raises issues related to providing methodological and computer support for

- Locating reusable assets
- Assessing their relevance to the current needs
- Adapting them to those needs.

The main goal of component-based software engineering is to decrease development time and development costs of software systems, by reusing prefabricated building blocks. Component-based software engineering (CBSE) aims at accelerating software development and decreasing development costs, by building software systems from prefabricated building blocks (components). Components are binary, independently deployable building blocks that can be composed by third parties. At the heart of CBSE lies software reuse, i.e., the use of existing artifacts for the construction of software.

It is widely accepted that software reuse is a major component of many software productivity improvement efforts, because it can result// in higher quality software at a lower cost and delivered within a shorter time period. Component is unit software that has business logics and interfaces, the component communicate with other components through its interfaces. Component- Based Development (CBD) approach develops software systems by assembling preexisting components under well-defined architecture or framework. The CBD approach brings high component reusability and easy maintainability, and reduces time-to-market. Therefore it improves productivity of software systems and lower development cost.

To implement a range of services in component based software, firstly a set of compatible components are identified from reusable component like a unit component. To reuse existing components, there are functional requirements that are summarized as following:

- Storing and Browsing of components,
- Easy access,
- Additional supports needed in reuse process.

To be collectively managed, the previously purchased and generated components should be stored and the components can be shown to let component re-users know the list of reusable components.

Each component should be available to identify uniquely and access conveniently at all times of access requests. The identification of reusable component should be known to determine which parts are to be partially or fully modified. Corresponding to the expected modification degree, the reusable components can be divided into the component to be newly generated and to be modified.

Since the internal interface of a component is communication method for interconnecting other component, the structural and functional information is

provided to check interface match degree and estimate reusable parts of a component.

To develop component-based software, requires various tasks such as search, selection, analysis, adaptation, deployment, test, assembly, purchase, development, and store components.

The CBD tasks are divided into two activities, one of which is the activity for component reuse and the other is the activity with component reuse. The for-component-reuse activity is composed of component purchase, development, and store tasks. To build component-based software, component-based software developers firstly search existing off-the-shelf components, and then collect reusable components. If they can't find reusable, components are purchased and newly generated. The newly generated components are required to be stored in the repository and be managed together. In the for component-reuse activity, are generated, which need to test the modified functionality to check correctness of the changes. In developing component-based software, the activities such as component identification, component modification, and component test make consequence for component reuse.

2. Purpose and Origin

Component-based software development (CBSD) focuses on building large software systems by integrating previously-existing software components. By enhancing the flexibility and maintainability of systems, this approach can potentially be used to reduce software development costs, assemble systems rapidly, and reduce the spiraling maintenance burden associated with the support and upgrade of large systems.

At the foundation of this approach is the assumption that certain parts of large software systems reappear with sufficient regularity that common parts should be written once, rather than many times, and that common systems should be assembled through reuse rather than rewritten over and over. CBSD embodies the "buy, don't build" philosophy espoused by Fred Brooks. CBSD is also referred to as component-based software engineering (CBSE).

Component-based systems encompass both commercial-off-the-shelf (COTS) products and components acquired through other means, such as non developmental items (NDIs). Developing component-based systems is becoming feasible due to the following:

1. the increase in the quality and variety of COTS products.
2. economic pressures to reduce system development and maintenance costs
3. the emergence of component integration technology
4. the increasing amount of existing software in organizations that can be reused in new systems.

CBSD shifts the development emphasis from programming software to composing software systems.

I.

3. TECHNICAL DETAILS

In CBSD, the notion of building a system by writing code has been replaced with building a system by assembling and integrating existing software components. In contrast to traditional development, where system integration is often the tail end of an implementation effort, component integration is the centerpiece of the approach; thus, implementation has given way to integration as the focus of system construction. Because of this, integrability is a key consideration in the decision whether to acquire, reuse, or build the components. Each activity is discussed in more detail in the following paragraphs

As shown in figure, four major activities characterize the component-based development approach:

1. Component Qualification
2. Component Adaptation
3. Assembling Components into Systems
4. System Evolution

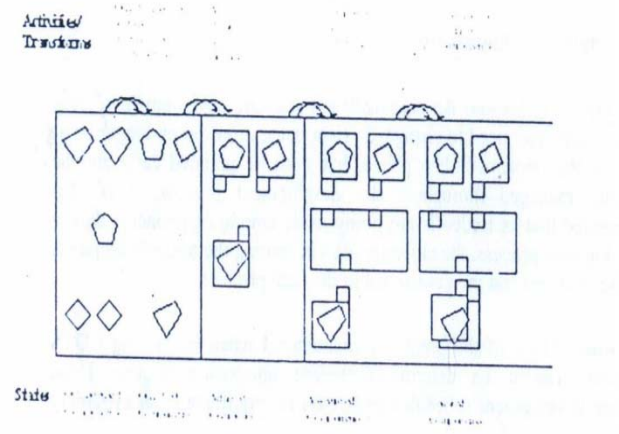

Figure 1: Activities of the Component-Based Development Approach

3.1 Component Qualification

Component qualification is a process of determining "fitness for use" of previously-developed components that are being applied in a new system context. Component qualification is also a process for selection components when a marketplace of competing products exists.

Qualification of a component can also extend to include qualification of the development process used to create and maintain it (for example, ensuring algorithms have been validated, and that rigorous code inspection have take place).This is most obvious in safety-critical applications, but can also reduce some of the attraction of using preexisting components.

There are two phases of component qualification: ***Discovery*** and ***Evaluation.*** In the discovery phase, the properties of a component are identified. Such priorities include component functionality (what services are provided) and other aspects of a component's interface (such as the use of standards). These properties also include quality aspects that are more difficult to isolate, such as component reliability, predictability, and usability. In some circumstances, it is also reasonable to discover "non-technical" component properties, such as the vendor's market share, past business performance, and process maturity of the component developer's organization.

Discovery is a difficult and ill-defined process, with much of the needed information being difficult to quantify and, in some cases, difficult to obtain. There are some relatively mature evaluation techniques for selecting from among a group of peer products. For example, the International Standards Organization (ISO) describes general criteria for product evaluation [ISO 91] while others describe techniques that take into account the needs of particular application domains [IEEE 93]. These evaluation approaches typically involve a combination of paper-based studies of the components, discussion with other users of those components, and hands-on benchmarking and prototyping.

One recent trend is toward a "product line" approach that is based on a reusable set of components that appear in a range of software products. This approach assumes that similar systems (e.g., most radar systems) have similar software architecture and that a majority of the required functionality is the same from one product to the next. The common functionality can therefore be provided by the same set of components thus simplifying the development and maintenance life cycle. Results of implementing this approach can be seen in two different efforts.

3.2 Component Adaptation

Because individual components are written to meet different requirements, and are based on differing assumptions about their context, components often must be adapted when used in a new system. Components must be adapted based on rules that ensure conflicts among components are minimized. The degree to which a component's internal structure is accessible suggests different approaches to adaptation:

1. ***White box,*** where access to source code allows a component to be significantly rewritten to operate with other components
2. ***Grey box,*** where source code of a component is not modified but the component provides its own extension language or application programming interface (API)
3. ***Black box,*** where only a binary executable form of the component is available and there is no extension language or API

Each of these adaptation approaches has its own positives and negatives; however, white box approaches, because they modify source code, can result in serious maintenance and evolution concerns in the long term. Wrapping, bridging, and mediating are specific programming techniques used to adapt grey-and black-box components.

3.3 Assembling Components into Systems

Components must be integrated through some well-defined infrastructure. This infrastructure provides the binding that forms a system from the disparate components. For example, in developing systems from COTS components, several architectural styles are possible:

1. ***Database,*** in which centralized control of all operational data is the key to all information sharing among components in the system.
2. ***Blackboard,*** in which data sharing among components is opportunistic, involving reduced levels of system overhead
3. ***Message bus,*** in which components have separate data stores coordinated through messages announcing changes among components
4. ***Object Request Broker (ORB) mediated,*** in which the ORB technology provides mechanisms for language-independent interface definition and object location and activation

Each style has its own particular strengths and weaknesses. Currently, most active research and product development is taking place in object request brokers (ORBs) conforming to the Common Object Request Broker Architecture (CORBA).

3.4 System Evolution

At first glance, component-based systems may seem relatively easy to evolve and upgrade since components are the unit of change. To repair an error, an updated component is swapped for its defective equivalent, treating components as plug-replaceable units. Similarly, when additional functionality is required, it is embodied in a new component that is added to the system. However, this is a highly simplistic (and optimistic) view of system evolution.

Replacement of one component with another is often a time-consuming and arduous task since the new component will never be identical to its predecessor and must be thoroughly tested, both in isolation and in combination with the rest of the system. Wrappers must typically be rewritten, and side-effects from changes must be found and assessed. One possible approach to remedying this problem is Simplex

4. Usage Considerations

Several items need to be considered when implementing component-based systems:

4.1 Short-term considerations

1. Development Process: An organization's software development process and philosophy may need to change. System integration can no longer be at the end of the implementation phase, but must be planned early and be continually managed throughout the development process. It is also recommended that as tradeoffs are being made among components during the development process, the rationale used in making the tradeoff decisions should be recorded and then evaluated in the final product.

2. Planning: Many of the problems encountered when integrating COTS components cannot be determined before integration begins. Thus, estimating development schedules and resource requirements is extremely difficult.

3. Requirements: When using a preexisting component, the component has been written to a preexisting, and possible unknown, set of requirements. In the best case, these requirements will be very general, and the system to be built will have requirements that either conform or can be made to conform to the preexisting general requirements. In the worst case, the component will have been written to requirements that conflict in some critical manner with those of the new system, and the system designer must choose whether using the existing component is viable at all.

4. Architecture: The selection of standards and components needs to have a sound architectural foundation, as this becomes the foundation for system evolution. This is especially important when migrating from a legacy system to a component-based system.

5. Standards: If an organization chooses to use the component-based system development approach and it also has the goal of making a system open, then interface standards need to come into play as criteria for component qualification. The degree to which a software component meets certain standards can greatly influence the interoperability and portability of a system.

6. Reuse of existing components: Component-based system development spotlights reusable components. However, even though organizations have increasing amounts of existing software that can be reused, most often some amount of reengineering must be accomplished on those components before they can be adapted to new systems.

7. Component Qualification: While there are several efforts focusing on component qualification, there is little agreement on which quality attributes or measures of a component are critical to its use in a component-based system. Useful work that begins to address this issue is:

1. "SAAM: A Method for Analyzing the Properties of Software Architecture".
2. Another technique addresses the complexity of component selection and provides a decision framework that supports multi-variable component selection analysis. Other approaches, such as:
3. The qualification process defined by the US Air Force PRISM program, emphasize "fitness for use" within specific application domains, as well as the primacy of integrability of components.
4. Another effort is Product Line Asset Support.

4.2 Long-term considerations

1. External dependencies/vendor-driven upgrade problem. An organization loses a certain amount of autonomy and acquires additional dependencies when integrating COTS components. COTS component producers frequently upgrade their components based on error reports, perceived market needs and competition, and product aesthetics DoD (Dept. Of Defense) systems typically change at a much slower rate and have very long lifetimes. An organization must manage its new functionality requirements to accommodate the direction in which a COTS product may be going. New component releases require a decision from the component-based system developer/integrator on whether to include the new component in the system. To answer "yes" implies facing an undetermined amount of rewriting of wrapper code and system testing. To answer "no" implies relying on older versions of components that may be behind the current state-of-the-art and may not be adequately supported by the COTS supplier. This is why the component-based system approach is sometimes considered a risk transfer and not a risk reduction approach.

2. System evolution/technology insertion: System evolution is not a simple plug-and-play approach. Replacing

one component often has rippling affects throughout the system, especially when many of the components in the system are black box components: the system's integrator does not know the details of how a component is built or will react in an interdependent environment. Further complicating the situation is that new versions of a component often require enhanced versions of other components or in some cases may be incompatible with existing components.

Over the long-term life of a system additional challenges arise, including inserting COTS components that correspond to new functionality (for example, changing to a completely new communications approach) and "consolidation engineering" wherein several components may be replaced by one "integrated" component. In such situations, maintaining external interface compatibility is very important, but internal data flows that previously existed must also be analyzed to determine if they are still needed.

5. Conclusion

It is widely assumed that the component-based software development approach, particularly in the sense of using COTS components, will be significantly less costly (i.e., shorter development cycles and lower development costs) than the traditional method of building systems "from scratch." In the case of using such components as databases and operating systems, this is almost certainly true. However, there is little data available concerning the relative costs of using the component-based approach and, as indicated in Usage Considerations, there are a number of new issues that must be considered. In addition, if integrating COTS components, an additional system development and maintenance cost will be negotiate, manage, and track licenses to ensure uninterrupted operation of the system. For example, a license expiring in the middle of a mission might have disastrous consequences.

References

[1] Jihyum Lee, Jinsam Kim, and Gyu-Sang Shin, "Facilitating Reuse of Software Components using Repository Technology", Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC'03), IEEE, 2003.

[2] Oliver Hummel and Colin Atkinson, "Extreme Harvesting: Test Driven Discovery and Reuse of Software Components", IEEE, 2004.

[3] www.sei.cmu.edu/str/descriptions/template

[4] Youwen Ouyang, Doris L. Carver, "Creation of Reusable Components Based on Formal Methods",

[5] Feather, M.S., Menzies, T., Connelly, J.R.," Relating Practitioner needs to research activities" in: RE 2003. International Conference on Requirements Engineering, IEEE Computer Society Press (2003) 352-361