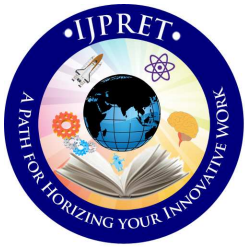
Research Article *ISSN: 2319-507X*
Sreenivasulu G, IJPRET, 2013; Volume 1(9): 41-56 *IJPRET*



INTERNATIONAL JOURNAL OF PURE AND APPLIED RESEARCH IN ENGINEERING AND TECHNOLOGY

A PATH FOR HORIZING YOUR INNOVATIVE WORK

OBJECT ORIENTED COMPONENT BASED DEVELOPMENT IN SOFTWARE ENGINEERING

G. SREENIVASULU, G. CHANDRA SEKHAR,

B. SUJANA, SK.NAZEER

IJPRET-QR CODE

PAPER-QR CODE

Quba College Of Engineering & Tech, Nellore

Accepted Date:

27/04/2013

Publish Date:

01/06/2013

Keywords

Component,
Mining,
Exploration,
Excogitation,
Exploitation,
Web Service,
Work Bench

Abstract

In the Software engineering context, reuse is an idea both old and new, programmers have reused ideas, abstractions, and process since the earliest days of Computing, but the early approach to reuse was ad hoc. Today, complex, high-quality computer based systems must be built in a very short time and demand a more organized approach to reuse. It encompasses two parallel engineering activities: domain engineering and component-based development. Domain engineering explores an application domain with the specific intent of finding functional, behavioral, and data components that are candidates for reuse.

Corresponding Author

Mr. G. Sreenivasulu

1. Introduction

Component reuse methodologies have been the recent focus of industry and academia alike, mainly driven by the increasing complexities of modern systems. Other major factors influencing this revolution are immense competition from competing vendors and consequently less time to market, the need for more open generic solutions of the Internet era, as opposed to the more closed solutions of the pre-Internet era and the need for developing solutions that can be easily verified are often referred to as design for verifiability.

This Dissertation describes about Components, Architecture, Design, Approach, Framework, Demands and Trends, Techniques and Towards Framework and Security in Component Based Development.

The focus of this paper is the issue of component matching for embedded systems, which are application specific reactive systems.

Information system evaluations are an area noted to be somewhat under-researched. Indeed, whilst much intellectual effort has been devoted to the development of information systems in a technology centric sense, the same does not seem to hold true for the evaluation of investment in such systems in a business-centric sense. Evidence suggests that the majority of investment decisions take place without a rigorous appraisal of the expected costs and organizational benefits and that they often represent and 'act of faith' based on competitive imperatives. Similarly, there is little evidence of evaluation during the operational lifecycle of the system.

This may be argued to be due largely to the problems of 'measurement', which can be described as follows. Firstly, the business costs and benefits associated with the development and use of an information system are inherently hard to understand the predict and, as a consequence, difficult to quantify and measure. Secondly, business organization is dynamic and changing and, as a consequence, business costs, benefits, risks and the like are a relative concept. Measurement and

evaluation thus need to be treated as an ongoing process. The pragmatic consequence of these points is that it may be argued that the perceived effort required for evaluation is assumed to be too great in the context of current business practice despite the high levels of investment. Given current levels of information system 'failure' and the cost associated with ongoing system maintenance, this assumption may be questioned. Despite the adoption of a methodical approach to system development, there is considerable evidence to suggest that information systems continue to take too long to build cost too much to implement and maintain and fail to meet the needs of their environment in the long-term. The dynamic nature of business organization may be argued to have much to do with these problems and, increasingly, the 'silver bullets' of system development attempt to address flexibility, the capability of the system to respond to the changing needs of the business environment in a timely and graceful manner.

2. Background

The goal of our research is to increase the possibility of finding and retrieving components that meet user's requirements. We have argued that improving the collaboration between component developers and users improves the ability to find suitable components. Though the collaboration of developers and users is not really new idea, there are very few real implementations that support collaboration. The assumption of collaboration is that components usually need to be modified to meet user's requirements. It is easier for the component developer(s) to achieve this because they have direct access to their products. Some promising technologies may be used at the developer's side to facilitate this customization, such as generative programming or software product line. Otherwise the customer may have to write significant glue code to use the component. The prerequisite of collaboration is to find satisfactory component candidates.

2.1. COMPONENT MINING PROCESS

The process of mining components and subsequently using them within an application domain can be divided into the three phases.

1. Exploration
2. Excogitation
3. Exploitation

These phases roughly correspond to the selection, specialization, and integration dimensions of typical software reuse methodologies. During the exploration phase, you elicit component requirements and – on the basis of component abstractions- select components. In this phase, the selected components and the system architecture determine your corresponding interface requirements. The excogitation phase deals with the encapsulation of the components that have to be mined and the implementation of suitable interfacing glue for connecting components with the rest of the system. The abstract nature of packaged components and interfaces means that many of them can be stored in a repository for future reuse or retrieved from this repository for direct reuse. Finally, during the exploitation phase, you use the reused and newly encapsulated components and corresponding interfaces to create a functioning system. The excogitation and exploitation phase are composed of three basic activities:

- Component encapsulation, where an existing stand-alone program is converted into a component object.
- Component glue implementation, where special-purpose components provide a uniform and reusable interfacing mechanism between the mined components and the rest of the system .
- Component use and composition, where component object are combined to form new structures and components.

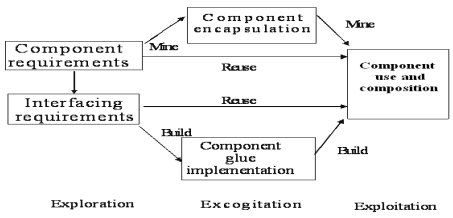


Figure 2.1: The Component Mining and Exploitation Process

2.2 COMPONENT BASED DEVELOPMENT

Software components have been talked about and implemented in varying forms for many years now. A component is a logical entity that solves a special purpose. Components are self-describing. The fixed interface to a Component describes enough to make it possible for clients to know how to use it. Component based development is the technique of using a Component Framework to develop Components.

Components are designed to be deployed within this framework. The framework is defined as an environment to:

- Create runtime instances of Components.
- Allow components to discover other Components.
- Allow Components to communicate with other Components.

Provide additional common services such as Persistence, Transactions, Location independence, Security, and Monitoring. CBD is primarily a technological advancement so the direct benefits it gives are technical. These technical benefits however lead to strong indirect business benefits. Some of the business and technical benefits are, Higher Quality product, Reduced time-to-market.

1) Reduced cost.
2) High reuse for future projects.
3) Complexity is managed Quality of solution
4) Independent design, implementing, and testing allow a high degree of concurrent development.

3. EXTENDING CBD WORKBENCH WITH WEB SERVICES

The Web service concept, is intended to provide a standard way of discovering and then using computer programs across the internet and therefore, there are no standard services to perform useful business functions or any consideration as how a service should be build, and there is no attempt to show how web services might be integrated into an existing application. The mechanism of service description is one of the key elements in Web Services architecture. The CBD

workbench having the producer, consumer subsystems and the component repository may be extended as a web service. Web services as defined earlier are a new generation middleware built on the XML technology. SOAP is a protocol fit for Internet. It is very promising and offers many great features that cannot be compared with other technologies. The producers and consumers can invoke the services of SOAP to implement the following in their programs:

1) Dynamically upload components

2) Search and download components

3) Provide dynamic plug-in capabilities to their applications and products

The consumers may search for a component based on the desired characteristics and functionality to be possessed by the component. The main web service searches the repository for the component based on the search criteria. When a match for the search criteria is found, then a web service is dynamically generated for that component and the functionalities desired by the consumer, then the UDDI registry is searched for the services that have the matching criteria based on the service description and functional description of each of the registered services. In this way, the resulting application may be a composition of components and calls to web services that in turn use the components to achiever certain functions.

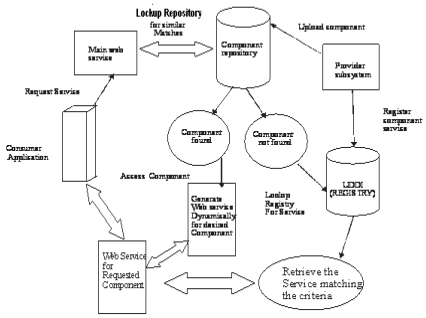


Figure 3: Architectural model

The application can be just a screen which offers a complete set of functions, but behind the screen, calls other web services to do the work. The functionality of the application may also be dynamically changed by the user, and the application would adapt to the new requirements and call the corresponding services.

4. TECHNIQUE FOR THE DESIGN OF COMPONENT-BASED APPLICATIONS

Component-based development (CBD) has become a much talked-about subject today. While the technology of CBD-as exemplified by environments such as EJB and COM has become increasingly mature, this has not been complemented by corresponding maturity on the methodology front. Of the few published methodologies available for the design of components, most address the process of building systems from a set fore-built component. However, an important scenario that is left un-addressed is that of the design of custom applications. In such contexts, the CBD design question involves the creation of business components from a set of software requirements. Currently there is no published prescription that addressed this need At best; practitioners rely on published collections of software patterns or heuristic guidelines such as those advocating correct component granularly. We, propose a comprehensive design methodology for indenturing two classes of business components based on an object oriented specification of requirements. These are entity components and process components. The methodology is illustrated with an example, and a real-life case study of an auction site is also described.

4.1 SUBTLE RELATIONSHIPS BETWEEN CBD AND 00 METHODOLOGIES

Much has been said and written about the similarity and differences between CBD and 0. For instance, a component –based system may or may not use 00 as the underlying development paradigm; however, given the popularity and maturity of 00 techniques, the use of 00 techniques as a basis for CBD is well accepted. The locus of our attention here is somewhat different. We wish to examine the process of transformation from a set of domain models to a set of implementation artifacts via a set of design artifacts. This process, is relatively well established in the 00 case, but merits some examination the case of CBD. We begin this examination by quickly running through some key ideas in 00. though many different 00 methodologies exist, in essence there are the phases of Requirements Gathering (RG) and Requirements Analysis (RA) or just Analysis, followed by one or more stages of Design.

As a representative methodology we take Jacobson's OOSE, whose key ideas now from part of Rationale's Unified Process [IO]. The models created during the Analysis stage can be viewed as first-cut at a design model [lo] and thus they form the basis of the design. For example, in Jacobson's OOSE, the Analysis stage yields three types of classes-entity, interface and control –which we may call early design objects. The design activity is a further refinement of these early design objects, during which consequences of the implementation environment are take into account. The phrase "early design objects" is important especially because it conveys the notion that these artifacts are the basis of the design yet abstract enough to be the basis for multiple implementation environments. With an implementation in an 00 language like C++, the translation of a set of early design objects to C++ code is known to be relatively straightforward.

Recent component development methodologies that have tried to address the methodological aspect of design include Catalysis, SCIPIO [121 and COMO [13]. Catalysis is specifically targeted as a method for component based development in which families of products are assembled from kits of components i.e., pre-built components. Catalysis is specifically targeted as a method for components based development in which families of products are assembled from kits of components i.e., pre-built components. Catalysis development process follows the RAD technique with stages of analysis, design, implementation and testing. It provides a systematic basis and process for the construction of precise models starting from requirements, for maintaining those models, for re-factoring them and extracting patterns and for reverse-engineering from detailed description to abstract models. The SCIPIO method incorporates business process modeling, workflow management and object-oriented analysis and design for the compo based development and evolution of open distributed business systems. The current system (As-Is :) is compared with the future system (To-Be) and controller transition from one to another. The aim is to reuse as much as is practical of the current legacy system and of publicly available

components. The SCIPIO method does not use the traditional OOAD approaches in the requirements phase. A technique more suited for the custom software development is COMO and bases its component development on unified process. The component development process consists of 4 phases: Domain analysis phase, component design, component build and component testing. The domain analysis phase results in identifying the components. These components are identified using two clustering techniques: use case clustering technique and use case by considering the ‹‹ *extent's* ›› relationship between the use cases. The second on addresses the clustering of the cohesive use cases and classes into components by applying a clustering algorithm. The clustering algorithm results in assigning the classes that have strong coupling with uses cases into components in which related use cases are contained. In the paper we address a key limitation of some of the earlier work- the lack of a comprehensive design methodology for identifying the components based on a specification of requirements. Rather than provide yet another end-to-end methodology like Catalysis, we address a very focused problem. Consequently, our methodology can be "plugged" into some of the end-to-end component methodologies for the purpose of component identification, especially in the case of custom software application development. Very briefly, our methodology starts with the familiar outputs of an object-oriented requirements specification activity –the domain object model and the use cases. These are subjected to detailed analysis to yield candidate sets of entities that can be clustered or grouped. This yields two sets of business components that we call entity components (EO) and process components (PCs) [14]. At a broad level the methodology is similar to COMO but there are important and significant differences, especially differences, especially in the clustering technique used to arrive at components.

We also bring out some subtle point that has to do with the granularity of components defined in our approach Vis-& Vis that of COMO. In our approach, unlike

in COMO, we retain the distinction between ECs and PCs. This allows us to better address environments such as EJB in which artifacts such as entity beans and session beans represent entity and process –related component functionality, and at the same time allow further clustering to suit environments in which COMO-style components are better suited –Entity.

The DOM forms part of the RG stage of most 00 methodologies such as [IO]. We state it here for the purpose of completeness. The DOM captures the structural aspects of the system by defining objects, their attributes, and the relationships (associations) between them in the analysis stage. In addition to specifying the names of the relationships (e.g. inheritance) that exist between objects, constraints on those relationships (e.g. multiplicity) are also given by an object model.

4.2 RISKS IN COMPONENT-BASED DEVELOPMENT

Risks in component-based development range from difficulties in establishing the quality of COTS software, to predicting how the components perform in a given context, to difficulties in composing applications, through to problems of managing component-based applications. The risks stem from four main factors, which are a consequence of the component ware paradigm: The black box nature of COTS software (i.e., 0 Lacks of information about the source of 0 the lack of component interoperability 0 the disparity in the customer-vendor these factors translate to technology and business risks. Business risks are associated with events that may result in loss of business through failure to deliver a system on time or within budget. Technology risks are related to the specific technologies used to build the system, for example, unreliable components. Technology and business risks are not mutually exclusive. Some risks may be related to both the business and technology. It is also important to note that business and technology risks sometimes oppose each other, for example, a business risk might be failure to meet a market window and a possible risk reduction strategy may be unknown design assumption Components.

These problems combine with poor component specification to: Diminish the quality of evaluation that can be done on a COTS component. More importantly it increases the potential for a component failing to interact with other system components. In a situation where many complex functions are replaced by a single large-scale integration COTS software this may have serious implications for exception handling and critical quality attributes (e.g. security, performance, safety etc). Components are packaged and delivered in many different forms (e.g. function libraries, off-the-shelf applications and frameworks). This may cause major difficulties during the integration process. Most COTS software is generally not tailor able or "plug and play". Significant effort is often required to build wrappers and the "glue" between components in order to evolve the application or tailor components to new situations. As the system evolves these wrappers may need to be maintained. There is a general lack of interoperability standard to facilitate the integration of components implemented using different component technologies. The use of COTS software introduces a vulnerability risk that may compromise system performance, security or safety. This is particularly critical for distributed systems and safety-related systems. The variability of specialized system domains (e.g. safety critical) and the competing nature of their quality attributes often make it difficult to adapt components to different application contexts without major modifications. It is important to note that COTS offering the similar functional may have very different system resource requirements (i.e., memory and processor requirements). This may adversely affect the system operation. The different customer-vendor evolution cycles may result in an uncertainty about how often COTS components in a system may have to be replaced and the extent of the impact of such a change on the rest of the system. This makes it difficult to plan and predict costs over the life cycle of a system. COTS component heterogeneity may result in complicated licensing arrangements in which no single vendor has complete control over the development artifacts for the purpose of evolution. Poor organizational quality standards related

change management might diminish the scope for evolving component –based systems. Upgrading to a new version of COTS software poses several risks: Hidden incompatibilities may cause unforeseen side effects in the: +tern necessitating a complete system update. 0 Changes in the quality attributes of a new version of COTS software (e.g. performance, security, safety, reliability etc.) may be incompatible with the user requirements. This may adversely affect the operational capabilities of the system. A new version of COTS software may provide additional undocumented capabilities. This may result in undesired side effects.

Stage	ID	Risk item
Requirements	R1	Lack of Cots-driven requirements engineering process. There is a general acknowledgement that good requirements engineering is essential for successful component-based system development [6] .However, few requirement methods address the problem of how requirements formulation process is affected by the availability of COTS software and how the user requirements are mapped to COTS products and component frameworks (see section 2)
Design	D1	The design assumptions of a COTS component are mostly unknown to the application builder. Also the perception of quality may vary across COTS software vendors and application domains. These problems combine with poor components specification to: • Diminish the quality of evaluation that can be done on a COTS component. More importantly it increases the potential for a component failing to interact with other system components. • In a situation where many complex functions are replaced by a single large-scale integration COTS software this may have serious implications for exception handling and critical quality attributes (e.g., security, performance, safety etc)

4.3 COMPONENT SECURITY – ISSUES AND AN APPROACH

Security vulnerabilities posed by third-party software components in Component Based Development (CBD) is a serious impediment to its adoption in areas that offer great economic potential, particularly in areas such as embedded software and large-scale

enterprise software. They raise questions about reliability and integrity of components, as well as the risks posed by any malicious code. This paper is a discussion of factors that affect component security and ways of assuring component security.

Component-based development (CBD) offers great promise in reducing software production costs through software reuse. Capacity to acquire and absorb new features, new technologies and experiential knowledge make components an ideal medium for encapsulating intellectual assets. Coupled with large scale applicability, components could thus bring about significant economic benefits, particularly in areas such as enterprise management and embedded software. A major technical challenge posed by CBD is the security and third-party software components, typically commercial-off-the shelf (COTS). The lack of security in component-based systems (CBS) may result in breaches of its own integrity, as well as of confidentiality and integrity of the underlying information assets. Component security has received some attention in literature, though, unsurprisingly, mostly in relation to operating systems. However, there are a few works devoted to CBS outside this area.

5. Future Work

Our next step is to populate the repository with embedded software components and related code templates. We are also developing a framework for code template based component composition. The framework interprets code template definition and manages code template instantiation based on NFR analysis results. It also generates glue code for buffer allocation and management and event delivery. For NFR analysis, the framework is designed to go through the nested templates recursively to compute the cumulative properties. In the first version of the framework, the use specifies specific component instantiations and component configuration parameter values to obtain the analysis results. Algorithms and tools will be developed to determine optimal instantiation and configuration.

6. Conclusion

Earlier drafts and prior works motivated and laid the groundwork for the integration of Web services and Component Based Development. Web services represent an evolution of the Web to allow applications to interact over the Internet in an open and flexible way. Interoperability between different systems is one of the primary reasons for using Web Services. Web Services are going to play an important role in the future of distributed computing, significant impacting application and system development. The various possibilities with which the components may be integrated well with applications have been explored and well exposed using web services. Building a component –based application can now be done starting from these composition patterns instead of the components. In practice this will be an iterative process where composition patens are selected based on available components and components are selected based on the specification given by the composition pattern. This process fits in the tradition of Software Development life cycles such as the waterfall mode, the iterative model and the spiral model. Composition patens can be viewed as a kind of use-case for the application.

The success of these systems on the market has been primarily the result of appropriate functionality and quality. Success in development, maintenance and continued improvement of the systems has been achieved by a careful architecture design, where the main principle is the reuse of components. The reuse orientation provides many advantages, but it also requires systematic approach in design planning, extensive development, support of a more complex maintenance process, and in general more consideration being given to components.

It is not certain that an otherwise successful development organization can succeed in the development of reusable components or products based on reusable components. The more a reusable component is developed, the more complex is the development process, and more support is required from the organization. Even when all these requirements are satisfied, it can

happen that there are unpredictable extra costs.

This paper discusses in some detail the major security issues affecting component security, in particular, the relevance of trust and reliability, how to related widely known existing security models to component security, the importance of component security modeling, particularly from the viewpoint of testing. It also outlines a CSP – oriented formal framework for modeling component security, demonstrating assignment of sensitivity labels and compartmentalization as envisaged in multi-level security architectures. Intended future research includes the consideration of type enforcement as a possible component security mechanism.

7. References

1. H. Change L. Cooke, M.Hunt.0. Martin .A. McNclly, and L. Todd, Surviving The SOC nonunion: Y guide to pldfom baaed design. Kluwer Academic, 1999.

2. P.C.Kanellakim and S.C. Smolka. CCS expreasiona, finite atate processes, and thrca Problem Of equivalence. Information and Comptrtaioa. 86.43-68. 1990.

3. N. Lynch and F. Vaandrager. Forward and backward simulations Part I: Untimedaystema. Information and Computation, 121 (2): 214-233. Sep.1996.

4. Mezini, M., Seiter, L. & Livebearer, K. Software Architectures and Component Technology: The Stute of the Art in Research and Practice. Kluwer.2000.

5. ezini, M., Seiter, L. & Livebearer, K. Software Architectures and Component Technology: The Stute of the Art in Research and Practice. Kluwer.2000.

6. Addison-Wesley, 1997.[3 J ITU-TS.ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, September 1993.

7. M.D. McElroy, Mass-produced Softhearted Components, In P. Nauru and B. Rendell,

8. Editors, Software Engineering, NATO Science committee, January 1969

9. Nierstrasz, S. Gibbs, and D. Tsichritzis, Componeni –oriented Sofhwu" Development, Communication of the ACM, 35 (9), 160-165, September 1992.

10. Altmann, J., and Pomberger, G.,"Cooperative Software Development: concepts, model,

11. And tools". In: Proceedings of the Technology of Object Oriented Languages and Systems, Santa Barbara, Califomia, pp. 194-277, August 1997.