Critical Issues in Component-Based Development

Rafael González & Miguel Torres
Departamento de Ingeniería de Sistemas
Pontificia Universidad Javeriana
Bogotá, Colombia
{ragonzalez, metorres}@javeriana.edu.co

ABSTRACT

Software development has been coupled with time and cost problems through history. This has motivated the search for flexible, trustworthy and time and cost-efficient development. In order to achieve this, software reuse appears fundamental and component-based development, the way towards reuse. This paper discusses the present state of component-based development and some of its critical issues for success, such as: the existence of adequate repositories, component integration within a software architecture and an adequate specification.

Keywords: Component Based Development, Component Based Software Engineering.

1. INTRODUCTION / PURPOSE

Typical problems in the development of computer-based information systems include: excessive customization (meaning, little or no reuse); complex integration and deployment; lack of coordination standards; lack of interoperability; difficulty in dealing with change; necessity of reducing time and development cost; demanding integration requirements; and lack of run-time flexibility [17]. Development of software has, however, continually evolved to deal with such challenges: from structures, to objects, to components. Regarding components, the body of knowledge and available tools already have enough drive and potential to allow Component-Based Development (CBD) to deal with the ever-present difficulties mentioned above. This paper discusses the present state of component-based software development and some of its critical issues, such as: the existence of repositories, component integration within a software architecture and an adequate ontology and specification.

The rest of this paper is structured as follows: Section 2 explains how components favor reuse, differentiating a component from an object. Section 3 summarizes the state of the art in component-based software development, presenting it as a new paradigm for software construction and showing how component repositories gain relevance in this context, along with the inclusion of components inside an architecture. Section 4 describes the key problem for CBD: component modeling and description; this is presented in terms of the need for an ontology of components, along with specification issues (static vs. dynamic and formal vs. informal) and description and modeling languages that can support the specification of the component’s ontology. In Section 5 conclusions are drawn and important issues discussed in the paper are outlined to direct attention towards researching component ontology as an unresolved issue in CBD.

2. FROM OBJECT ORIENTED DEVELOPMENT TO COMPONENT REUSE

In this section we will describe what a component is and how it is different from an object (or class). After this, we will mention some of the advantages and issues of component-based development in terms of software reuse.

What is a Component?

The definition of a component is ample and potentially ambiguous, which is why, for software, the ontology of a component remains an open and critical issue towards the development of guidelines and practical efforts in component-based software development. From a logical perspective, a component is a way to model real-world concepts in a computer system’s domain. This allows for decomposition of complex problems into entities, processes or transactions, for instance. From a “physical” (in the software sense) perspective, components are independent units of software, which implement logical abstractions. Thus, a component is any coherent design unit which may be packaged, sold, stored, assigned to a person or team (for development), maintained and, most importantly, reused [21]. Another cryptic and recursive definition of component is the one provided by Councill et al [11]: “A software component is a software element that conforms to a components model and can be independently deployed and composed without modification according to a composition standard”. This understanding poses the question of distinguishing between components and the classic “object” concept.

From Objects to Components

The notions of instance, identity and encapsulation are associated with the “object” notion, whose properties are: being an instantiated unit with unique identity; having an externally observable state; and encapsulating its state and behavior. A component, on the other hand, is an independent deployment unit, built by a third party, which has no externally observable state [26].

Since the origins of software development, the idea behind libraries, subroutines, and abstract data types, was modularization. In the 80’s and 90’s, the arrival of the Object Oriented (OO) paradigm brought about the possibility of creating highly encapsulated and easily maintainable systems. In OO, a component was normally seen as a collection of related classes which provided a consistent and logical set of services [22]. For some authors, the fact that CBD is a natural evolution of OO [7][24], gives origin to Object-Oriented Components, which includes benefits such as: interoperability,
extensibility, reusability, easy assembly, flexibility in run-time, integrated standards for design, flexible development, speed, quality and reliability when using components-off-the-shelf (COTS) [17].

However, it seems that the OO paradigm has not proven to be useful enough for reuse, because most OO applications are developed from scratch. In contrast, while the OO paradigm is based on programming techniques and models, CBD extends such ideas to other areas, which complement the programming field. For example, to be able to achieve an effective interaction among components, it is necessary to adopt rigorous design and documentation disciplines, and modeling standards. This is the basic notion of Design-by-Contract, a discipline conceived within the OO paradigm, which fits more naturally inside the CBD approach [5].

CBD Implications on Reuse

It has been established that the most important advantage of CBD is the possibility of reuse it brings into the software development field. The main idea is to be able to build systems based on reliable and already tested components as it is done in other engineering disciplines [3]. To achieve this, it is important to extend our comprehension from a software understanding (where models describe software technically and code is the focus) to a component understanding (where the functionality, application and adaptation requirements of components gain importance) [2]. This means that besides a model which technically describes the software, the component must be viewed and understood in terms of its reusability (functionality, requirements and restrictions). When this is fully achieved, it will be possible to build systems by means of component integration and by directing programming efforts to the integration and not to coding functionality.

CBD is the means to exploit software reuse, which offers robustness, reliability and interoperability, but these benefits must outweigh the cost of achieving reuse and it seems that CBD is not there yet. Some argue that building a system from scratch may still be cheaper than building a mostly reused fully component-based system [23]. Aside from the adaptation cost, components usually have an undesired performance cost also (performance is usually proportional to customization), resulting in software that, in most cases, performs less efficiently than one fully customized from scratch [5]. Another difficulty in achieving successful reuse is that the original developer of a component usually makes implicit assumptions with regards to the possible applications of his components, making these uses implicit and hard to find by other developers attempting to reuse the component [12].

It can be concluded that reuse is not a direct benefit of CBD; for it to be effective, it must be coupled with an adequate component’s description that aims toward reducing adaptation costs and that will allow improving performance, without affecting significantly its interoperability or increasing its development cost. One way to achieve this is by using a component’s formal specification; such specification should include the component’s static restrictions (consistency) and dynamic restrictions (allowed sequences of execution, allowed redefinition). It would be naive to believe that there already exists a component in a repository satisfying in its entirety the requirements of a given problem; thus, the component must be adapted to the problem and to the system’s architecture, so it can be used in that context. There doesn’t seem to be a shortcut regarding adaptation of reusable components.

3. COMPONENT-BASED DEVELOPMENT

This section establishes component-based development as a new paradigm of software construction and the success factors required, such as the existence of an adequate component repository and the placement of components within the context of architectures, patterns and structures.

CBD as a new Software Development Paradigm

Traditional information systems development is based on large work groups and long periods of time, which has resulted in undesirable economical and chronological consequences, disrupting an organization’s competitiveness. The biggest challenge for developing successful software systems then, will be to build systems in a shorter lapse of time, with lower cost and inside a changing and complex environment [8]. This also suggests the need for a more permanent solution, which will bring about the possibility of information systems development that may be flexible and adaptable to an organization’s conditions and existing technology. Research and practice point towards the CBD paradigm as the technological means to achieve this goal [1][3][8][15][29]. This approach includes improvements in: quality, throughput, performance, reliability and interoperability, reducing development, documentation, maintenance and staff training time and cost [18]. Most recent trends in software engineering show that future developments will follow in the CBD path. This argument is partially confirmed by the large amount of component development technologies that exist today (CORBA, EJB, DCOM, and .NET among others), and also given the amount of components (COTS) available in the market [2].

Although CBD promises to improve the software development processes, quality, productivity and reuse in particular [28]; these achievements are not new in other areas of industry [3]. The biggest contribution of the Industrial Revolution was precisely this. From that point forward this paradigm has been used in the electronics, automotive, and construction industries. Automobile parts, hardware components (such as video and networking cards), and construction design patterns are reused in every new project, making use of rigorous and reliable standards, without losing innovation in the process (as the dynamics of these particular sectors had shown).

In the last few years Component Based Software Engineering (CBSE) has emerged as a new paradigm for plug-and-play software, where components are provided and stored in component repositories which provide the interface information for each component. This approach supports complex and usually distributed applications, while reducing maintenance costs and increasing reliability [21].

This new form of applying Software Engineering requires particular aspects for its success, such as: component and architecture selection; adaptation and integration of components inside the chosen architecture; and maintenance of the components along with the evolution of requirements [2][3][16]. This new approach might not be easy to implement, because of the fact that it must guarantee coexistence and compatibility among different component versions, and from
different sources. So in order to be able to maintain independence, which facilitates maintenance of third party components, it is recommended to divide CBSE into two levels: the component level and the application level [5]. This means that in one level, the engineering of the components itself is the focus, while in the second level, the application engineering is the focus (without, of course, disconnecting these two aspects).

CBSE remains as an immature discipline [3] that can learn from other engineering areas, and from the experience of the object-oriented paradigm and traditional software engineering, but for it to be successful, it must be used along with an adequate support by a component repository.

**Component Repositories**

In order to make CBD more cost-effective and allow components to be more easily found, good repositories must be available. Thus, the research field in component retrieval is highly dynamic. The aim is to obtain effective, easy to use (navigate), complete and efficient repositories. However, a problem arises from using open (on-line) repositories: it must be guaranteed that components comply with the minimum security expectations [20]. An integrator or architect will expect components developed by third-parties to have the properties (behavior) which the creators claim, along with maintaining the basic requirements of quality, legitimacy, abstraction, encapsulation, low coupling and high cohesion.

From this, it is gathered that a certifying authority is required; one which should aid in the following: 1) outsourcing (managing the outsourcing contract for the development of components and auditing the performance of the developing institution); 2) component selection (selection of the most adequate components according to user requirements, such as functionality, security, trustworthiness and performance); and 3) testing of components (verifying that components satisfy the requirements with acceptable quality and reliability) [9].

Aside from certifying conformance, it is useful that the certifying authority guard the relevant security aspects, studying components in the following manner [20]: 1) characterizing atomic components independently; 2) checking the compatibility of security features among components; and 3) validating property visibility with respect to external entities. Some additional efforts would indeed be desirable, such as the implementation or adoption of security standards, such as the Department of Defense’s Orange Book, the NIST Common Criteria or ISO/IEC’s 15408 standard. An example of a repository that both validates a component’s properties and compliance to standards is the CLARiFi project [6][10].

**Architecture, Structure and Patterns**

Another critical issue is the place of components within an architecture with the aid of design patterns. In this context, architectures are a set of decisions regarding the platform, component frameworks and interoperability design among such frameworks [26]. In recent years, software development has oriented its focus upwards, in the direction of an abstract level of architecture specification [5]. The transition towards CBD has introduced new distributed technological infrastructures based on Microsoft’s COM (Component Object Model), OMG’s CORBA (Common Object Request Broker Architecture) or Sun Microsystems’s EJB (Entreprise Java Beans). When these models are coupled to a well-defined layered architecture, they become the best technological support to develop the infrastructure necessary for component-based systems [17].

Besides software architectures, design patterns exist to optimize object-oriented systems design, based on a catalogue of generic structures guided towards solving recurring problems. Some see design patterns as micro-architectures [26]. Yau and Dong [28] propose the use the use of design patterns for integrating components in CBD. After designers have selected a design pattern to describe relationships among components (within a particular system), the pattern must be instantiated; this instantiation consists in transforming the participating relationships into design interactions. After this, the structure and interactions are verified, to guarantee that the pattern’s restrictions are complied with, and wrappers can then be employed as decorators of the components. In this way, architectures and patterns are formally integrated into the system’s design.

From the previous section, one could say that with component-based software development, the engineering process centers its attention on high-level design through architecture and design patterns, in which components are assembled and adapted to this design.

4. **COMPONENT DESCRIPTION AND MODELING**

One final issue, crucial for successful development of component-based software is specification. A component is usually specified (described) in terms of its interface (as Interface Definition Languages do), but this offers no information regarding the component’s performance, security or synchronization [12]. There are three different options for specification: it may be informal, formal or semiformal. Informal specification concentrates on describing the component through natural language; formal specification presents more specific aspects such as semantic information or requirements in formal logic; semiformal approaches describe the components without being either totally formal or totally unstructured.

**Component ontology**

In the field of software components, an ontology may be thought of as the semantic definition of components (including its behavior and non-functional properties). Such semantics has two dimensions: knowledge of the component’s domain and knowledge of the component software [25]. Domain knowledge provides the basic structure (motherboard) over which the product’s knowledge is specified [3]. Frequently, this domain information guides component modeling [ibid.: 362]; however, it is focused on non-functional properties (performance, security, ...) with little attention to user-oriented descriptions (access conditions and instantiation, for instance) [12].

It could be assumed that formal descriptions of a “white box” type could solve this problem, providing enough information for the use and understanding of the component, but the effort required to use and comprehend this formal models dissuades developers from using it, inclining them towards “black box” type descriptions [2][27]. Because of this, a more appropriate model would be one that offers not only understanding of the domain (requirements vs. capabilities), but also of the program
itself (interfaces, data types, syntax, parameters, and acceptable ranges) and the situation (structure, connections, and flows) [2].

Although an ontology (in the traditional computing sense) tends to be associated with the definition of a context (or vocabulary); in this case, we refer not only to the context, but also to the conditions and internal characteristics of components that must be specified.

Component Specification

Given a component’s definition and how it may be integrated into a structure or architecture, it is also necessary to provide a specification of the component for it to be easily identifiable and usable. At first, the component may be specified in terms of its interface and its basic calling procedure (such is the case of interface definition languages – IDL), but this type of specification may be poor in terms of describing additional (yet very relevant) information like performance, security and synchronization [12].

There are two different ways of specifying a component, not just in terms of its interface: 1) informal specification, which centers on using natural language for describing the characteristics of a component and its interface; and 2) formal or semiformal specification, which presents information through more rigorous and unambiguous language [12][21][23].

One semiformal proposal is LI PS [12] which, besides use policies, shows instance and thread support information. Another semiformal approach is CDM (Component Description Manager) [21] that presents a classification framework for the effective management of components, based on the problem domain, semantic information and the component’s ontology. The resulting classification tree concentrates on characteristics, grammar, components and standards. Such classifications are useful in different contexts: for instance, in component brokers or repositories, they help in finding and identifying components; in a development environment they help in verifying compatibility and performance of the components.

Static vs. Dynamic Specification

Aside from the level of formalism, a component may also be specified statically or dynamically. The static structure of a component is described in terms of the services that the component provides and which remain valid for any instance of the component; such specification, however, makes it difficult to describe certain restrictions [5]. Dynamic structure may describe legitimate call sequences and behavior of the services or operations of the component in a given moment of the component’s execution (i.e. in run-time) [22]. An example of this type of dynamic description is provided by the Abstract State Machine Language (AsmL) based on Microsoft’s .NET technology, which incorporates non-determinism ad transactions to generate IL and verify a component in run-time [4]. Another approach consists in defining the component in a mathematical and hierarchical fashion, based on finite state-machine modeling of classes and the components which contain them. [22].

Some description and modeling languages

Associated with the type of specification available for a component is the language chosen to specify it. Such languages may be formal, semiformal or formal to answer to the required type of specification, but they must also be standard languages in order to stimulate adoption, comprehension and interoperability.

The UML is an obvious natural candidate for describing components given its widespread adoption and generic nature. Its strengths are (among others) that it integrates in a natural way, various modeling approaches (disconnected in the past) and thus becomes a stronger, more mature and integrated alternative; it also offers enough popularity and available tools to make it attractive in practice. However, some argue that it is still ambiguous for describing components and that its understanding of a component is limited [14]. For instance, valid redefinition of operations is a restriction that is not formally modeled with UML [22]. Nonetheless, the use of UML within a set of object-oriented practices and languages, make it one of the best modeling languages for designing (visually, in particular) systems and describing components [19].

One particular effort which extends the UML for components is the Catalyst approach [13]. CBD and Catalyst are clear examples of architecture description languages which support analysis and design of components [21]. Catalyst offers a non-ambiguous definition of component and subsystem interfaces to ensure adherence to a given model and set of business rules. Indeed, this means that Catalyst allows the description of a component according to the component and application levels presented in section 3.1 [2]. Its disadvantage is that any representation of a component with Catalyst means that the component is built with the Catalysts approach, making it difficult to describe an already made component or adopting a different development method.

One last language worth pointing out is Microsoft Research’s AsmL (already mentioned), an executable specification language that provides the possibility of verifying components statically and monitoring behavior dynamically.

One could also think of using architecture description languages (ADLs) as possible languages for specifying components. Although they are aimed at architectures, these are made up (basically) of components and connectors. Although their high level of abstraction and limited commercial adoption might make them less attractive, the focus of CBD towards architectures and patterns makes ADLs desirable for representing components within structures making their relationships and roles explicit.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have established some of the advantages and challenges associated with component-based development. At first, we clarified the difference between components and objects and how the first are more suitable for software reuse, through flexibility and reliability. We have also pointed out some critical issues for effective CBD: the existence of component repositories is one of the most important requirements; the existence of conditions and guarantees of security is also fundamental; having components supported by and integrated within structures and high-level architectures is also seen as a very desirable approach.

But the issue that is of highest relevance in this paper is component specification. Without adequate specification,
The suggested direction for the component software research paper points towards research in component ontology and a changing environment with many restrictions. In the end, this technologies as an excellent tool for building robust software in a changing environment with many restrictions. The suggested direction for the component software research community is thus the study of component ontology that may result in the definition of hierarchy, granularity and specification language considerations (perhaps connecting components with a particular application domain).

6. REFERENCES


[27]. Weinreich, R.; Sameinger, J., “Component Models and Component Services: concepts and principles,” Heineman,
