Guidelines for Designing Reusable Software Components

Basem Y. Alkazemi
Collage of Computer and Information Systems
Umm Al-Qura University
Makkah
Saudi Arabia
Email: bykazemi {at} uqu.edu.sa

Mohammed K. Nour
Collage of Computer and Information Systems
Umm Al-Qura University
Makkah
Saudi Arabia

Abd-El-Kader Sahraoui
CNRS, LAAS,7 avenue du Colonel Roche;
Univ de Toulouse, UTM, LAAS, F-31100 Toulouse
France

***Abstract*— Software component reuse is of obvious importance to the software engineering process and is increasing in prominence in enterprise software development nowadays. However, standard practices for designing reusable software components are lacking in many software development houses, as their reuse activities are either done in an ad-hoc manner at the very small scale or as part of a software production line. This paper discusses the key characteristics of reusable components and proposes guidelines for deriving the design of reusable software components.**

Keywords--- software repository, software components, interface, design, verification.

I. INTRODUCTION

Component reuse is an old paradigm that has been commonly exploited in many different professions. In car assembly lines, for example, motors, body parts and many other components are reused from one model to another. Rarely are new parts built from scratch. Electronic engineers assemble their integrated circuits from resistors, transistors, diodes and many other reusable components. They simply search for the required component on the corresponding data sheets that explain the detailed specification of each type of component so that they can reuse them.

In software, the concept of software reuse has existed since the beginning of programming, as programmers reuse algorithms, sub-routines and pieces of code from previously created programs. The idea of reuse in software was first formalized by McIlory [13], who emphasized the need to componentize software systems. So, applying McIlory's idea led to thoughts about building software systems in a similar manner to building hardware systems (e.g. electronic circuits). Later on, more advanced research work emerged that discussed reuse and its possible directions, emphasizing the significance of reuse [14,16]. Nowadays, reuse has become one of the standard paradigms that most leading software development vendors, such as HP, IBM and Motorola, practise in their production lines, and many others have reported successful experiences with applying reuse in their software development projects, such as the examples provided in the C.R.U.I.S.E. book [1].

Software reuse is a process in which organizations describe a set of systematic operations to generate, organize and locate reusable components for future development. When software reuse is discussed, two main techniques are commonly recognized, namely, developing with reuse and developing for reuse. The act of classifying and searching for software components belongs to the former technique, while the act of designing and developing components is the core of the latter technique. In fact, development for reuse is a prerequisite for development with reuse, as one cannot reuse a component if it is not available in the first place. However, a commonly accepted standard for designing reusable software components seems to be unrecognized widely until now. We believe that most of the work is done either in an ad-hoc manner for in-house development or as part of software production lines for enterprise-level development. Thus, our main focus in this paper is to discuss the complexity of development for reuse and to propose guidelines for potential directions towards standardizing this technique.

II. SOFTWARE COMPONENTS

'Software component' is a term that has various definitions in the literature, in that there is no single accepted definition of the term yet available. The following descriptions are the most prominent ones within the software industry. Brown and Wallnau [2] described components as nearly independent and replaceable parts of a system that satisfy some functionality in the context of a well-defined architecture. The component can be bound dynamically and accessed through a well-defined interface at run-time. Szyperski et al. [3] described a software component as a unit of composition with a specified interface and explicit context dependencies. The component can be deployed independently and subject to composition by a third party. Meyer [4] described a software component as a software element that can be used by other software elements (e.g. clients), possesses an official usage description and is not tied to any fixed set of clients. Heineman and Councill [5] described a component as a software element that conforms to a component model and can be deployed independently and composed according to composition standards without modifications. Yang and Ward [6] described a component as a coherent and configurable package that is available independently of the application in which it has been used and with a well-defined interface that can be used in

different contexts to interact and communicate with other components to form a system. Brown and Short [7] characterized a component as "…an independently deliverable set of reusable services". Hopkins [8] described a component as a physical package of executable code that exhibits a well-defined interface.

Our view of software components is that components are just parts that fit into a system in order to extend its functionality. They must exhibit characteristics through their interfaces to facilitate incorporation into systems and also for identifying them for reuse. We have adopted a very general model of the terms 'system' and 'component': a software system is composed of a number of software components, each of which may of course be a system in its own right; and a system may subsequently be used as a component in another system. A system defines a number of characteristics that it requires components to match. Components exhibit a number of characteristics by which they can be identified as reusable candidates in a system. A component might be complex or atomic. A complex component is one that is composed of a number of sub-components, while an atomic component is one that cannot be decomposed any further into smaller components. Sub-components represent the internal dependencies that a component needs to work (e.g. a custom library). A component can be considered reusable to a system developer only if it provides the required functionality expected by the developer; otherwise it will be of no use to them. The functional specifications of components are described via their corresponding interfaces, for example, a Java interface or an XML file as in web services in the form of WSDL. We consider this type of interface as the *functional interface* of a software component. It simply tells us what services a component can provide. There is another type that is very important to component reusers in order to make sure the found component can be deployed and work correctly in their system. We refer to this type as the *architectural interface* of a software component [19]. This interface tells us how to get at a component's functionality. A simple example of part of an architectural interface for a source code component is the programming language in which it is written; a C++ class will not fit into Java source code due to the differences in their formats. Another characteristic might be whether a Java class is thread safe or not. Indeed, object-oriented languages such as Java provide a particularly rich environment for software component (i.e. class) reuse, with conventions being defined for assisting class reuse. JavaBeans and Enterprise JavaBeans are examples of a component's architectural types (i.e. component models), and the definition of what is actually required in a component is the information to be identified in an architectural interface. An architectural type defines the values of the characteristics identified by an architectural interface that, if matched by a component to what a system requires, then the component can fit architecturally into that system. Figure 1 illustrates a fine-grained view of the system model describing the relationship between an architectural type and an architectural interface.

In Java, the language features of 'interface' and 'abstract class' can be used to define (and check) conformance to a particular architectural interface, although naming conventions are also used. So, if a system requires components to implement a method called "`public void run()`", then all components must define this method in order to fit into that system. Reusers can refine their search criteria by providing the definition of the architectural type that their system requires. For example, if a developer wants to reuse a "parser" component that fits into an EJB-based system, then part of the search criteria can be refined by providing the necessary lifecycle methods that an EJB architectural type defines.

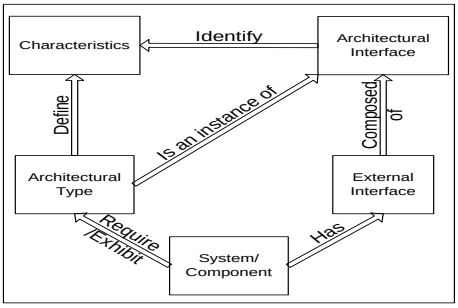

Figure 1. Fine-grained ontology of the system model

III. SOFTWARE PRODUCT LINES

The software product lines (SPLs) approach has become the de-facto standard nowadays among the software engineering community as the most effective way to practise software reuse [9]. The idea of software product lines was mainly derived from the need to develop several software products that share some common behaviour. Organizations that are practising the product line approach have identified two development roles:

- Domain engineering: The term 'domain' is used to denote or group a set of systems or functional areas that exhibit similar functionality. It is concerned with the development and maintenance of the shared components across a product line.
- Application engineering: This area is concerned with the development of products in the product line using the shared components, or component libraries [15].

In fact, domain engineering activities are linked to the development for reuse technique, while application engineering is related to the development with reuse technique discussed earlier.

Typically, every organization that applies the product line approach should have their own repositories that store the desired reusable components that belong to their development context, as this can reduce the time spent searching for a component and make it easier to locate the desired components. Also, because the developers know exactly how their repository is structured, they can know precisely how to find what they want. However, there might be some occasions where developers cannot find the desired components in their repository. So, they could develop them from scratch or purchase them from external vendors, and then populate them in their repository for future reuse.

IV. SERVICE ORIENTED ARCHITECTURE

Service-oriented architecture (SOA) is an architectural style whereby software components are deployed as services.

Theoretically, the main distinguishing characteristics between components and services in the context of SOA are in the types of interfaces they exhibit. Components define two types of interfaces, as we described in the previous section: functional and architectural. Services in SOA are functional units that respond to requests regardless of any architectural considerations, assuming that the architectural interfaces are fixed among an organization or a production line. So, services communicate with each other via a predefined protocol of interaction, such as JMS, SOAP, IIOP or RPC, whereas components might comply with different architectural characteristics than that of a system, interacting successfully through middleware.

In many software development organizations, the SOA style is favoured over component-based development due to its flexibility to integrate various types of systems by means of an enterprise service bus (ESB) [10]. However, this approach has its own drawbacks (as described in [11]) with respect to the services register, discovery, binding and execution. We are going to give broad guidelines for designing reusable components in general, regardless of the SOA consideration, as we believe services are a sub-set of components.

V. DESIGN CONSIDERATIONS FOR BUILDING REUSABLE COMPONENTS

As we described in section 3, a reusable component is one that provides the required functional interface needed by a developer. In order to reuse the component smoothly, the component must comply with the architectural interface required by the system to be developed. Based on that consideration, we categorize the guidelines into functional and architectural characteristics. We will provide our proposed guidelines on the fly while discussing the different characteristics.

The functional characteristics that a designer needs to consider when building reusable components are:

- ***Generality***: It is commonly known that component functionality is the key driver that influences any reuse activity. More general functionality might lead to more potential for reuse in a wider range of problem domains. For example, a spell checker is a parameterized general purpose component that can be plugged into many word processers. However, the relationship between generality and reusability is not always proportional, as too much generality might require unnecessary business logic to be incorporated into the system to be developed, which may negatively impact the execution performance. So, there is a trade-off between component generality and the degree of reusability, which the component designer needs to decide upon. We believe that general components should represent the common business logic identified from domain engineering activity. Thus, designers can decide whether a component is reusable in one problem domain or might be reused across different domains.
- ***Granularity****:* Component granularity can range from fine-grained to coarse-grained components. In its simplest form, a component can be represented as a simple method or procedure that can be reused in a specific programming language. A more advanced form of a component can be denoted in classes and packages of classes, whereby an entire set of classes might be reused as libraries for system development. Still, this form of components is restricted for reuse in the scope of programming languages with some exceptions to Java, where it can communicate with non-Java libraries through JNI. A further more complex component can be represented as an entire framework, whereby a developer can reuse and extend its functionality. This is the most common type of reuse nowadays among software development organizations; it is very rarely found that an enterprise system is built from scratch. The most complex form of components' complexity is represented in an entire application that might be reused and customized to fit business needs. This practice is common among solution providers, e.g. Oracle, Microsoft and other large corporations. These companies provide solutions to customers and customize their systems as per their requirements. The common levels of granularity are illustrated in Figure 2 as layers of potential components. A component designer needs to identify the level of granularity their component has in order to limit or widen its reuse possibility. We believe that overly fine-grained components might not reflect the correct architecture of a system, as the small components represent the design. Moreover, network traffic might be considerably increased. An overly coarse-grained component, on the other hand, might greatly impact the complexity of the component and, subsequently, increase the maintenance overhead.

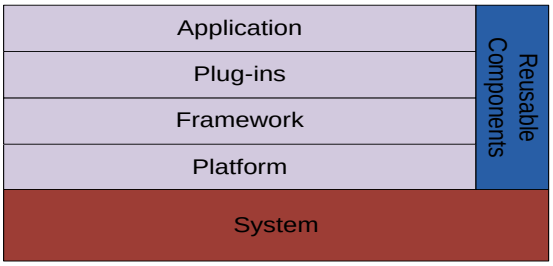

Figure 2. Scope of reusable components

On the other hand, the architectural characteristics that a designer needs to consider when developing reusable components are:

- ***State***: Software components have different states that they can go through during their execution. Each component model defines different interfaces to manipulate their respective states. For example, Java Applet must implement the following methods:

```
Public void init();
Public void start();
Public void stop();
Public void destroy();
```

These methods define the different states that an Applet component can have. Thus, a component designer needs to define a required interface that captures the complete

lifecycle of their component's states and needs to describe the interface explicitly in the attached documentation.

- ***Entry point:*** This is commonly known among the software architecture community as a "port". The entry point is the first block of code that should be invoked to initialize a component. Some components (i.e. class-based components) may provide special methods that must be executed to provide initialization, while others (i.e. framework-based components) may require the presence of special tools or files for their initialization. For example, a standalone Java application must have a method called "`public static void main()`" to be initialized, while an Android plug-in can be initialized by reading a file called "`plugin.xml`" with the presence of a method called "`public abstract int getItemId()`". So, a component designer must decide about the type of component needed (a class-based or framework-based component) and consequently define their entry point.

- ***External dependencies:*** A software system may require its composing components to use dependencies that it provides for them to fit into the system. For instance, a Java system requires its composing components (i.e. Java classes) to use a library called "`java.io`" to achieve the basic input and output functionality. Also, an Android system requires its components (i.e. plug-ins) to use a plug-in called "`org.Android.osgi`" to allow the system to control their execution. So, components must use the external dependencies that are provided by a system in order to be integrated successfully into the system. A component designer must define the dependencies that should be packaged and delivered with the component itself and also explicitly define the required dependencies to be provided by the system by means of an interface.

- ***Data exchanging model:*** After a component is initialized, it will be ready to receive data for processing and sending out. The mechanism of handling data must be defined according to the requirement of the system under development in order to avoid potential mismatches. For example, a component that receives data via parameters may not fit into a system that requires their components to read data input from a file. Both the system and the components must agree upon a data exchanging model. So, a component that employs the push model will not fit into a system that assumes its components exchange data according to the pull model. Therefore, a designer must precisely define an interface for describing the data exchanging model of the component.

- ***Control type:*** The way control is exchanged can differ from one component to another. One component may synchronize its execution with a system, so the component can return control to the system upon the completion of its execution. Another component might execute asynchronously with the system. Thus, identifying the different mechanisms of control flow is necessary for reusing components successfully in a system. It is the role of a component designer to define the control type.

These characteristics capture the most significant design considerations that component designers need to address. There are some usability aspects that can also contribute to component reusability. However, we believe that a complete and correct documentation manual can be sufficient to enhance component usability and comprehensibility. Other reusability facilitators such as security are also of importance to consider in order to gain reusers' trust. Component security is significant in the context of COTS components, which reusers might need to purchase from vendors. However, we believe this characteristic is not a big concern within a single production line as it should be established as part of the security standards; hence it is omitted in this paper.

VI. REPOSITORY SYSTEM FOR REUSABLE COMPONENTS

In order to standardize the development of reusable components, we have proposed a prototype of a repository system to automate the development of components' architectural interfaces that represent only one dimension to be considered when developing reusable components, as discussed earlier. Figure 3 depicts a prototype of a repository system design to support the development of reusable components. The repository system is composed of several components, such as refactoring tools, classification scheme, matching tools and database storage. The core elements of the repository system design that directly affect the development of architectural interfaces are the classification scheme and the refactoring tool. The matching tools are, somewhat, supporting mechanisms.

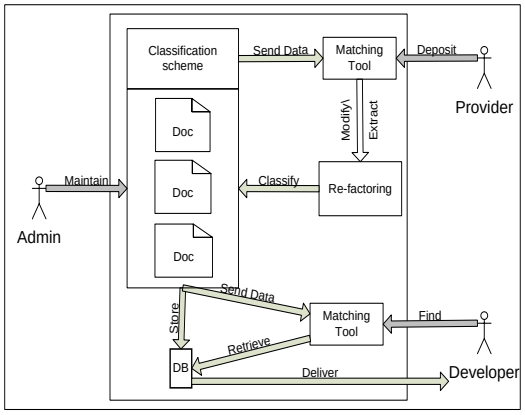

Figure 3. Prototype of repository system design

The repository system should support three different views:

1. The provider's view: This view is concerned with the source code provider – a developer or another repository system.
2. The developer's view: This view is concerned with matching, or refactoring, what is inside the repository to fit the reuser's needs, based on satisfying the required architectural interface.
3. The admin's view: This view is concerned with ensuring the flexibility of the whole system by allowing new categories

(i.e. user-defined architectures) to be added to the classification scheme

Briefly, the envisaged operation of the repository is as follows. When a software system is deposited in the repository, the characteristics of its components are identified and matched against the definitions of various architectural types available in the classification scheme. Hence, the architectural types defined by the admin in the classification scheme will permit that software to be automatically analysed to identify its architectural type and hence be classified. The overall architecture of that deposited system can be identified; various components within the system will also be identified. As a simple example, the classes within a Java application might be identified as conforming to the *Java Session Bean* architectural type and can be identified and classified as such. A developer can search for components based on the architectural type definition that his system requires together with free-text searches for functionality. Searching for a Java Session Bean component can show all of those in the repository, but the repository may also be able to refactor other components to offer to the developer. For example, a simple Java class could have the required Session Bean methods added (or a wrapper class formed) by the refactoring tool. Although such a refactored class would not be complete and might require additional modifications by the developer, this would still offer a more complete solution to the reuser's needs than the current repositories.

It is important that a classification scheme is not closed and that developers (and providers) can identify and define new architectural types to be supported by the repository. For example, the MVC (model-view-controller) design pattern is often informally used in appropriate applications. A repository user might wish to define the architectural types of these parts to permit components to be interchanged. Utilizing the concept of architectural interfaces in the design of the repository will permit such interfaces to be defined by the user.

VII. PROVISIONAL EXPERIMENTATION

The notion of an architectural interface represents a key aspect in the design of reusable components. As a result, our initial evaluation was concerned with examining the soundness of the architectural interface. We have limited our study to the Applet architectural type to examine the soundness of the approach at this stage. The experiment aimed at examining whether components can be identified as conforming to the Applet architectural type based on an XML description provided for the Applet architectural type. We developed a prototype of an XML-based specification language called ArchInt [12] to represent components architectural characteristics. At this early stage, the ArchInt specification captures only method signatures, dependencies (external/internal), and inheritance relationships. We applied ArchInt to define the Applet architectural type in this experiment, as displayed partially in Figure 4.

```
<ArchInt>
  <name>Applet</name>
  <uses_ArchInt>
      <name>Java Class</name>
  </uses_ArchInt>
  <must__have>
      <Method>
        <name>setStub</name>
        <param>
          <string>AppletStub</string>
        </param>
        <returnType>void</returnType>
      </Method>
      .
      .
    </must__have>
</ArchInt>
```

Figure 4. Partial listing of Applet architectural type specification

We have built a tool called *ArchIntParse* in Java language to automate the checking of software components against an architectural description. About 500 random instances of Java components were selected from Sourceforge.net to be examined by the tool.

All the components were examined by the *ArchIntParse* tool and the results were as follows. The tool identified 23 components as conforming to the Applet architectural type, while the remaining components did not. In order to evaluate the validity of this result, all the 23 components were inspected by hand in order to examine if they really satisfied the Applet characteristics. We found that all the 23 components did conform to the Applet specification. We also examined the remaining components that our tool had not identified as Applets and we found that 6 components satisfied the specification of the Applet architectural type and the remaining 471 components were not Applets at all. This result indicates that our approach can successfully identify components if they exactly match the full characteristics of an architectural type. The experiment also revealed that partial conformance is not supported by the current architectural interface paradigm, and that was the reason these components were not identified by the tool.

VIII. CONCLUSION

Reusable components are valuable assets that can considerably reduce development costs and time to market. However, finding appropriate reusable components is one of the key hindrances to exercising reuse. This is attributed to the lack of standard design practices that help designers to not only focus on the functional aspects of their components but also the architectural aspects, in order to enhance the reusability of their components. We proposed a set of guidelines that touch upon the principal design considerations for building reusable software components. The proposed guidelines were then applied to check some of the characteristics of software components in order to identify them as potential reusable candidates. Apparently, the verification process of software components has been done successfully. We

believe the verification can be generalized further to cover components certification for real-time systems [18].

Our planned future work is to utilize the repository system design to develop a general purpose verification model that is not limited only to code artefact but can also consider traceability models in order to define reusable contexts. The essence of the approach is to enable verifying components compatibility from requirements to implementations through a significant traceability model [17]. The traceability will not be restricted to simple syntactic matching but also components' semantic considering system context as additional verification criteria, and hence verification ensures the coverage of various artefacts from requirements to code components.

REFERENCES

[1] Almeida, E., Alvaro, A., Garcia, V., Mascena, J., Burégio, V., Nascimento, L., Lucrédio, D. and Meira, S., 2007, *C.R.U.I.S.E. – Component Reuse in Software Engineering* , C.E.S.A.R. e-book.

[2] Brown, A. and Wallnau, K., 1998, The Current State of CBSE. *IEEE Software*, 15(5): pp. 37-46.

[3] Szyperski, C., Gruntz, D. and Murer, M., 2002, *Component Software – Beyond Object- Oriented Programming*. 2nd edition, Addison-Wesley (ACM Press),

[4] Meyer, B., 2003, The Grand Challenge of Trusted Components, *in 25th International Conference on Software Engineering (ICSE '03)*, IEEE Computer Society.

[5] Heineman, G. and Councill, W., 2001, *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley,.

[6] Yang, H. and Ward, M., 2003, *Successful Evolution of Software Systems*, Artech House Publishers,

[7] Brown, A. and Short, K., 1997, On Components and Objects: The Foundations of Component-Based Development, *in Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97)*., IEEE Computer Society.

[8] Hopkins, J., 2000, Component Primer. *Communications of the ACM*.. 43(10): pp. 27-30.

[9] Bosch, J., 2001, Software Product Lines: Organizational Alternatives, *in Proceedings of the 23rd International Conference on Software Engineering*, IEEE Computer Society Press. Nov., pp. 91-100.

[10] Chappel, D., 2004, *Enterprise Service Bus: Theory in Practice*, O'Reilly Media, 2004.

[11] Zhu, H., 2005, Building reusable components with service-oriented architectures. *Information Reuse and Integration Conf*., pp. 96-101.

[12] Alkazemi, B.Y., 2011, A Precise Characterization of Software Component Interfaces, *Journal of Software(JSW)*, Mar, pp. 349-365.

[13] McIlroy, 1968, M.,Mass Produced Software Components. *In Software Engineering: Report on a Conference by the NATO Science Committee*.NATO Science Affairs Division, pp.138-150.

[14] Mili, H.; Mili, F.; Mili, A., 1995, Reusing software: issues and research directions, *IEEE Transactions on Software Engineering*, Aug, pp.528-562.

[15] Frakes, W.B.; Pole, T.P., 1994, An empirical study of representation methods for reusable software components, *IEEE Transactions on Software Engineering*, Aug, pp.617-630.

[16] Frakes ,W. B., and Kang ,K. C., 2005, Software reuse research: status and future, *IEEE Transactions on Software Engineering*, Aug, pp. 529-536.

[17] Sahraoui, A.E.K, 2005, Requirements Traceability Issues: Generic Model, Methodology And Formal Basis, *International Journal of Information Technology and Decision Making*, Jan, pp.59-80.

[18] Krichen , M. and Tripakis, S., 2009, Conformance Testing for Real-Time Systems. In Formal Methods in System Design, June, pp. 238-304.

[19] Alkazemi, B.Y., 2009, *Exploiting the Architectural Characteristics of Software Components to Improve Software Re-use*, PhD thesis, School of Computing Science, (Newcastle University, U.K)