Software Component Specification Using Design by Contract

Yi Liu and H. Conrad Cunningham
Department of Computer and Information Science
University of Mississippi
237 Kinard Hall
University, MS 38677 USA
(662) 915-5358
{liuyi,hcc}@cs.olemiss.edu

Abstract

This paper describes methods for identifying appropriate software components for an application and for specifying the components' operations rigorously. It uses the theory and methods of the design by contract approach for specification of the functionality. The actual implementations of a component's operations are hidden from the clients and encapsulated within the component. A component communicates with another component only through one of the other component's supported interfaces. Hence, a component can be easily replaced by another that implements the same operations. By using design by contract, we build reliable reusable components.

1. Introduction

Most contemporary software systems exist in highly dynamic environments. Their requirements change frequently and they must be built or modified on challenging development schedules. The software systems are decentralized. They execute in a distributed fashion and the development and maintenance of the software and data involved may be distributed among many groups in the enterprise. In this context, software systems need to be modular and easy to change; the development methods and technologies should support reuse of analysis, design, and tested code.

Increasingly, the software development community is approaching the development of such distributed, enterprise-level software applications with component-based methods and technologies. The idea is that a new software application can be built quickly and reliably by assembling preexisting frameworks and components with a few new components. Furthermore, it should be possible to handle changes in the requirements by replacing a small number of components. For component-based development to be successful in an organization, the software developers must give close attention to the design of components as independent abstractions with well-specified behaviors.

What is a component? According to Szyperski [1998], "a software component is a unit of composition with a contractually specified interface and explicit context dependencies only." Furthermore, it "can be deployed independently and is subject to composition by the third parties."

A component can be represented as shown in the diagram in Figure 1 [Fleisch 1999]. A component's internal design and implementation are strongly encapsulated and it exclusively communicates with other components through its interfaces. The interfaces provide inputs and outputs that can be connected with other components. That leads to inter-component

dependencies being restricted to individual interfaces rather than encompassing the whole component specification (or worse yet, particular implementations of the specification). Thus, one component may replace another even if it has a different internal implementation, as long as its specification includes the interfaces that the client component requires. This allows a component to be replaced by another with minimal impact on the clients of that component.

This paper examines how design by contract techniques [Meyer 1992] [Mitchell 2001] can be used to provide the precise specifications needed to enable the components to be replaceable and reusable. Following the general approach proposed by Cheesman and Daniels [2001], the paper uses the Unified Modeling Language (UML) [Fowler 1999] to express structural relationships and its associated Object Constraint Language (OCL) [Warmer 1999] to express the design contracts.

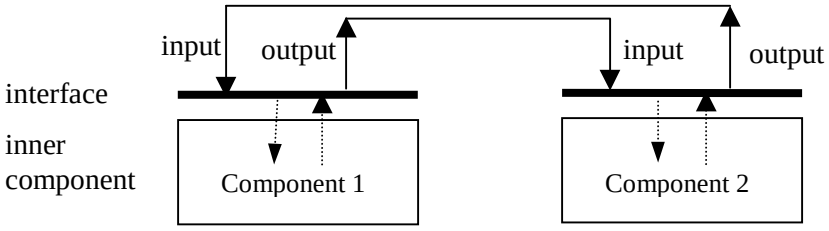

Figure 1: Components and Their Interconnections

2. Interfaces

The first phase of the design process is the analysis of the requirements. In this process, we construct a ***use case model*** and a ***domain model*** for the application [Cheesman 2001]. The use case model captures the user's expectations for interactions with the system [Fleisch 1999]. The domain model defines the relevant business concepts and how they relate to each other

Next, we move into design by refining the domain model to build a ***business type model***. That is, we identify those business concepts (i.e., types) that are within the scope of the software system to be built. A key step for component-based development is the identification of the ***core types***. Each core type is a set of functionality and data around which a component interface is defined. Each core type is selected because it has an independent business identifier and is relatively decoupled from other core types. Because the core types are independent of other each other, the corresponding interfaces are also independent of each other.

Figure 2 shows the interface responsibility diagram that is produced from business type model. We have identified two core types, `CoreType1` and `CoreType2`, and two supporting types, `Type1` and `Type2`. We associate a component interface with each core type. Each interface will include operations that enable manipulation of the instances of the core type it manages.

By analyzing the relationships among the types, we must then assign the responsibility for each supporting type to an interface. In Figure 2, for example, we determine that the looser connection is between `Type1` and `CoreType2`. Thus we assign `Interface1` to manage the types `CoreType1`, `Type1`, and `Type2` and assign `Interface2` to manage `CoreType2`, which does not have any other associated types.

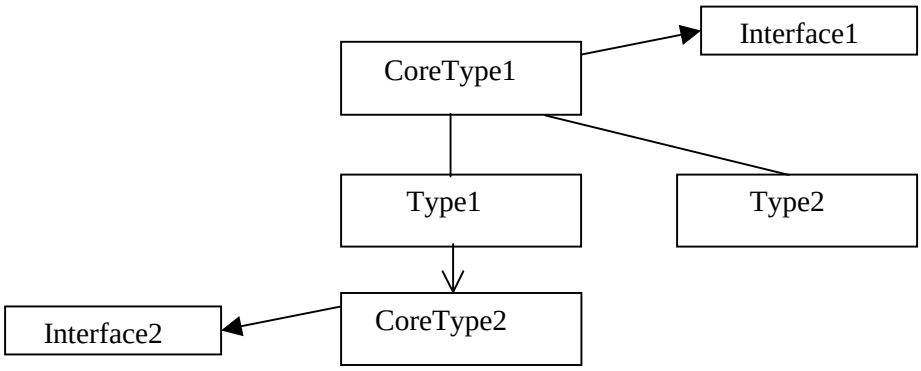

Figure 2: Interface Responsibility Diagram

From the interface responsibility diagram, we now build an ***interface information model*** for each interface. Each interface information model contains the information types needed to completely specify the behavior of that interface's operations. Some of the types included may actually be managed by other interfaces.

In Figure 2, for example, `Interface1` needs to use types `CoreType1`, `Type1`, `Type2` and `CoreType2`. However, the `Type1-CoreType2` association brings a dependency. The rule used here is: when a type owned by one interface refers to a type owned by another, the referenced type appears in the interface information models of both interfaces. So, there will be a type called `CoreType2` in both the `Interface1` and `Interface2` models. The attributes of `Coretype2` in the different models may be somewhat different because these two types have separate and distinct roles. The interface information model for `Interface1` is shown in Figure 3 represented as a UML class diagram.

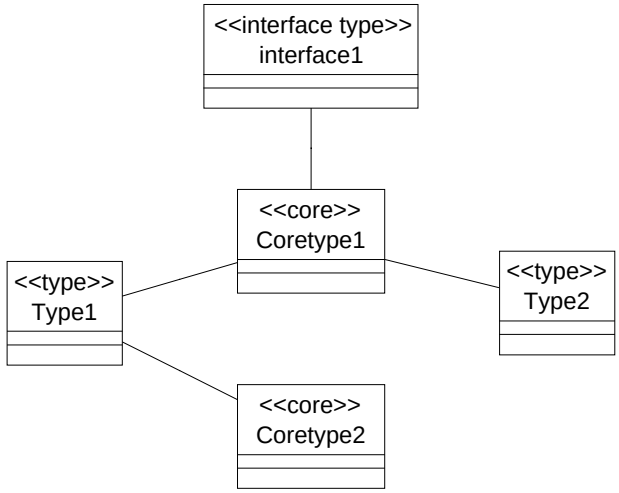

Figure 3: Interface Information Model for Interface1

Now, the information types of an interface information model are given in the same specification package as the interface itself. Thus, interfaces are distinct and independent from each other. Initially we associate a component with each interface that we have identified above. Later we may combine these small components to form larger ones that will support multiple interfaces.

3. Design by Contract

As mentioned above, development of pluggable components is a key motivation of our approach. That is, it should be possible to understand precisely what a component does based on the specification for the operations in its interfaces. It should be possible to replace one component by another that implements the same set of interfaces. It should thus be possible to reuse a component reliably with several different clients in different contexts.

Design by contract is a design approach developed by Meyer [1992]. It is used here to provide precise specifications for the functionality of components and to enhance their reliability. According to Meyer, a ***contract*** is a collection of assertions that describe precisely what each feature of the component does and does not do. The key assertions in the design by contract technique are of three types: invariants, preconditions, and postconditions.

An ***invariant*** is a constraint attached to type that must be held true for all instances of the type whenever an operation is not being performed on the instance. In the case of our component methods, we can attach invariants to an interface to specify properties of the component objects that implement the interface. For example, an invariant might state that the value of some attribute is always greater than zero.

Preconditions and postconditions are assertions attached to an operation of a type. A ***precondition*** expresses requirements that any call of the operation must satisfy if it is to be correct. A ***postcondition*** expresses properties that are ensured in return by the execution of the call. In our approach the precondition gives the contractual requirements on the client (that is, caller) of the interface and the postcondition gives the corresponding contractual requirement on the supplier of the operation, the component object that implements the interface. For example, an operation to delete a record from a collection might have a precondition requiring that a record with that key exists and a postcondition requiring that it no longer be an element of the collection.

Assertions are logical expressions about the entities in an interface's information model. They give component developers a precise description of the behavior of a component implementing the interface. A component can only be considered as implementing an interface if all instances of the component satisfy all the assertions in the interface's contract. (If a component implements an extended form of an interface, then it also implements the base interface.)

Contracts are important in providing support for pluggability and reuse of components. If a set of interfaces F of a component is needed in an application, then any component that implements all interfaces in the set F may be used, that is, may be plugged into the application

Assertions can also help improve the reliability of a component. They are checked at runtime to help test and debug the implementation. A precondition violation indicates a bug in the client. The client did not observe the conditions imposed on correct calls. A postcondition or invariant violation is a bug in the supplier. The supplier failed to deliver on its promises.

4. Course Registration System Example

Here we use as an example a course registration system for a college. With this system, a student may register for classes. Once given access, the student may select a term and then build a personal class schedule from among the classes offered that term. A student may add and delete classes from the schedule. The system passes the information about the student's schedule to the tuition billing system. An instructor may use the registration system to print a listing of the

students in his or her class. The administrator may maintain student and instructor lists and course information.

After analyzing the requirements, we specify the domain model and the use case model. Then, we refine the domain model into a business type model. From the business type model, we get the interface responsibility diagram represented by the UML class diagram shown in Figure 4.

In this example, we identify `Person`, `Course`, and `Term` as the core types. The corresponding interfaces are thus `IPersonMgt`, `ITermMgt`, and `ICourseMgt`, as shown in Figure 4. We assign types `Person`, `Student`, `Instructor`, `Student_Schedule`, and `Instructor_Schedule` to the `IPersonMgt` interface, assign types `Term` and `Section` to the `ITermMgt` inteface, and assign type `Course` to the `ICoureMgt` inteface.

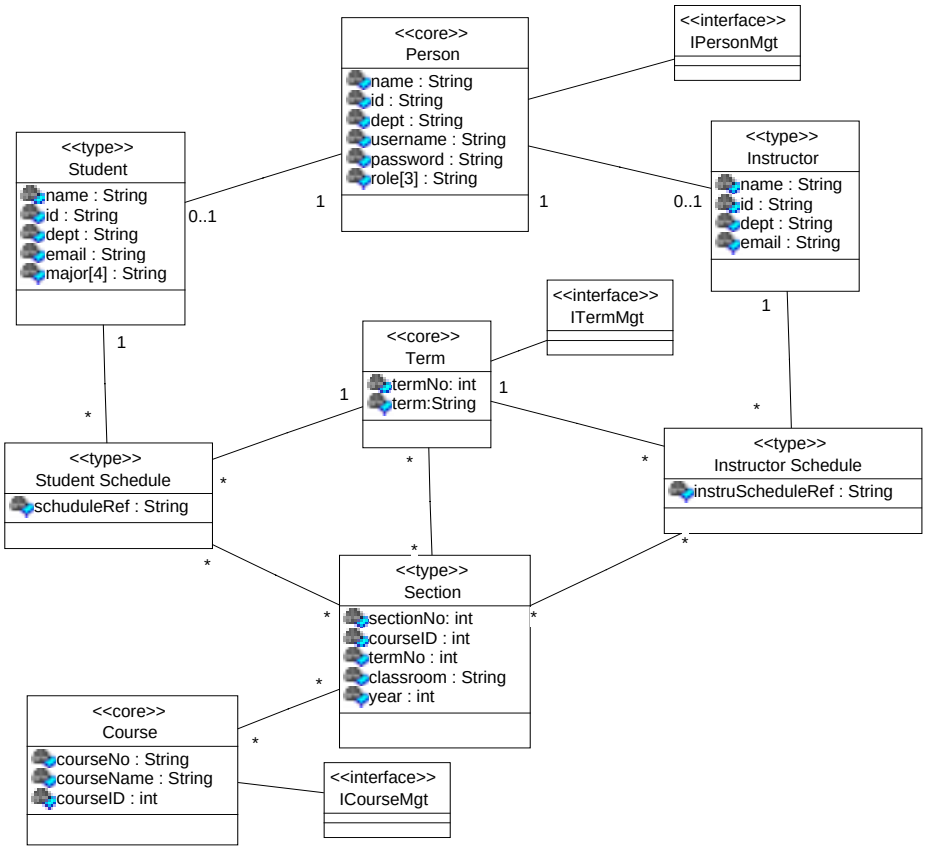

Figure 4: Interface Responsibility Diagram for Course Registration System

We use the interface `IPersonMgt` to illustrate the use of the design by contract approach. The interface information model for `IPersonMgt` (shown in Figure 5) needs to contain types for `Person`, `Instructor`, `Student`, `StudentSchedule` and `InstructorSchedule`. Because types `Term`, `Section` and `Course` provide important information for `StudentSchedule` and `InstrutorSchedule`, they should also be included in this interface information model. Type `Term` also should be included in the interface information model for `ITermMgt` and type `Course` for `ICourseMgt`. The interface models we build will be independent, but they will require some communication.

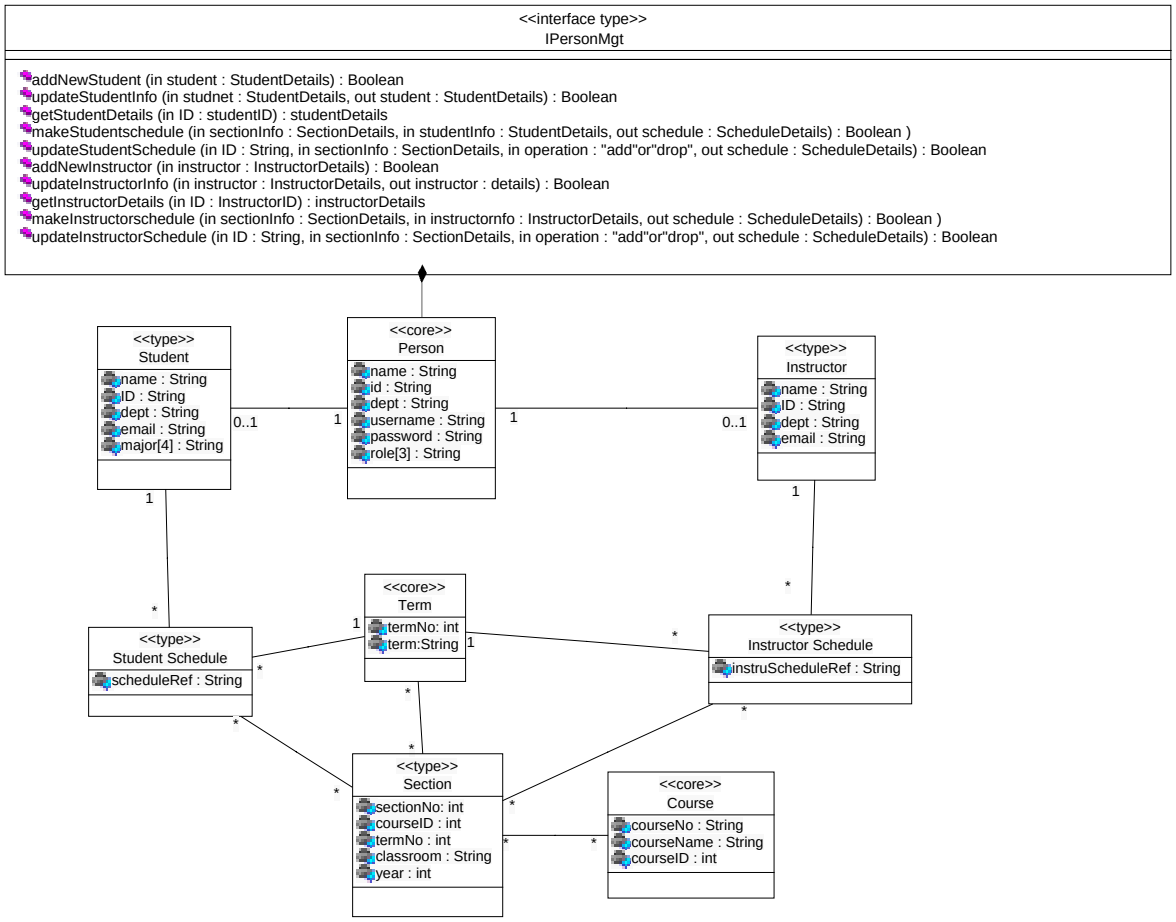

Figure 5: Interface Information Model for the IPersonMgt Interface

Consider the `makeStudentSchedule` operation of the `IPersonMgt` interface as an example design contract. This operation creates a new student schedule and adds a single course section to it.

We use the Object Constraint Language (OCL) to write the precondition and postcondition constraints. In the precondition, we require the student and section information provided as arguments to be valid. In the postcondition, we ensure that an appropriate result is achieved. On a successful execution of the operation, we ensure that the new student schedule has a unique schedule reference and that the schedule created is for the correct student and includes the desired course section. The information returned about the new schedule matches the information retained in the system

Below is the full operation specification written in OCL. Note that the specification we provide for this operation says only ***what*** must be achieved in terms of an abstract model of the component object's state. It does not say ***how*** the effect of the operation is to be achieved. The designers and implementers of the component are free to implement it in any convenient manner. The various implementations can thus potentially replace each other.

IPersonMgt::makeStudentSchedule(in sectioninfo:sectionDetails, in studentinfo:studentDetails, out schedule:scheduleDetails):Boolean

Pre:

```
----------------section and student information are valid
Course ->exists(c|c.id = sectioninfo.courseId)     and
Term->exists(t|t.termNo = sectioninfo.termNo)   and
Section -> exists (se|se.sectionNo = sectioninfo.sectionNo)  and
Person->exists(z|id = studentinfo.studentID) and
Student ->exists(y|id = studentinfo.studentID)
```

Post:

```
Result implies
   StudentSchedule@pre->forall(x|x.scheduleRef <> schedule.scheduleRef)
   and
   let s = ( StudentSchedule – StudentSchedule@pre)->asSequence->first in
      s.schedule.scheduleRef = schedule.scheduleRef and
      s.schedule.id = schedule.id   and
      schedule.id = studentInfo.studentID   and
      s.schedule.section = schedule.section    and
      schedule.section = sectioninfo.section
```

5. Discussion

Component technology standards such Sun's Enterprise JavaBeans (EJB), Microsoft's COM+, and the Object Management Group's CORBA Component Model provide good frameworks for defining and deploying distributed, server-side components. The techniques sketched here are generally applicable to development using any of the technology standards.

A combination of a component standard such as EJB and the specific signatures of the operations in the interfaces describe the shapes of the sockets for plugging-in components. Design contracts serve two complementary purposes. First, they can be used to specify the expected functionality provided by any component plugged-in to a particular socket. Second, they can be used to specify the actual functionality provided by a component. The design contracts provide a means for checking whether a component provides the expected functionality.

The design by contract methods (using UML and OCL) can also be used to rigorously specify other types of components. They are useful for precise definition of the functional aspects of client-side and Web components. They, of course, can be used in the design of object-oriented programs, the context for which the approach was originally articulated by Meyer.

Design by contract methods are based on the theory and techniques of formal specification and verification going back to the 1960s. The foundations are solid and rigorous, but formal methods have never had wide-scale acceptance in the practitioner community. With the design by contract approach, Meyer has sought to bring formal methods into wider use within the object-oriented programming community.

D'Souza and Wills' Catalysis [1999] and other such methods have extended formal techniques into the realm of component-based systems and frameworks. The UML Components approach of Cheesman and Daniels [2001] seeks to simplify the complex, but powerful, Catalysis methods

and make them more widely accessible. The work in this paper is based in the UML Components approach.

The authors are interested in methods for building component frameworks with plug-points for components at several different levels, including in the user interface. For this purpose, the UML Components approach has several shortcomings. It results in component systems with relatively flat structures because it does not support nested components or intermediate-level reuse structures such as frameworks. In addition, it does not support specification of user interface or user dialog components. Recent methods such as KobrA [Atkinson 2002] deal with such issues more effectively, but at the cost of greater complexity.

6. Conclusion

This paper describes methods for identifying appropriate component interfaces for an application and for specifying the operations in those interfaces rigorously. It uses the theory and methods of the design by contract approach for specification of the functionality.

The resulting interfaces are independent of each other. The actual implementation of an interface's operations are hidden from the clients and encapsulated within the component. A component communicates with another component only through one of the other components supported interfaces. Hence, a component can be easily replaced by another that implements the same interfaces. By using design by contract, we build reliable reusable components.

7. Acknowledgements

This work was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)." It was also supported by the Department of Computer and Information Science at the University of Mississippi.

8. References

[1] C. Atkinson, J. Bayer, et al. *Component-based Product Line Engineering with UML* , Addison Wesley, 2002.

[2] J. Cheesman and J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software***,** Addison Wesley, 2001.

[3] D. F. D'Souza and A.C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach,* Addison Wesley, 1999.

[4] W. Fleisch. "Applying Use Cases for the Requirements Validation of Component-Based Real Time Software," *Proceedings of the Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE, 1999.

[5] M. Fowler and K. Scott. *UML Distilled*, Second Edition, Addison Wesley, 1999.

[6] B. Meyer. "Applying Design by Contract," *Computer*, IEEE, October 1992, pages 40- 51.

[7] R. Mitchell and J. McKim. *Design by Contract, by Example*, Addison Wesley, 2001.

[8] C. Szyperski. *Component Software – Beyond Object-oriented Programming,* Addison Wesley, 1998.

[9] J. B.Warmer and A. G. Kleppe. *The Object Constraint Language – Precise Modeling with UML,* Addison Wesley, 1998.