Chapter 7

Role-Based Component Engineering

Jilles van Gurp, Jan Bosch

Introduction

COTS (Commercials-Off-The-Shelf) components have been a long-standing software engineering dream [1]. Several technologies have tried to fulfill this dream but most of them, so far, have failed, even if there have been some successes (e.g. visual basic components). From time to time, a promising new technique appears. The most successful technique to date has been Object Orientation (OO), but even this technique has failed to deliver truly reusable COTS components. In this chapter we investigate a promising extension of OO (i.e. role-based component engineering). Role-based component engineering extends the traditional OO component paradigm to provide a more natural support for modeling component collaborations.

The idea of role-based components is that the public interface is split into smaller interfaces which model different roles. Users of a component can communicate with the component through the smaller role interfaces instead of using the full interface. In this chapter we will examine why roles are useful, discuss methods and techniques for using them and discuss how they can be used to make better OO Frameworks.

Role-based components

Four different definitions of a component are given in [2] and also a number of definitions have been discussed earlier in this book. This indicates that it is difficult to get the software community as a whole to agree on a single definition. Rather than continuing this discussion here, we will focus on aspects of object-oriented components that are relevant to role based component engineering (For a more elaborate discussion of these concepts, see [3]):

- *The interface.* The interface of a component defines the syntax of how to use a component. The semantics of the interface are usually implicit (despite efforts to provide semantics in various languages (e.g. Eiffel [4]).
- *The size or granularity.* One purpose of using components is to extend reusability, so the larger the component the more code is reused. A second purpose of components is to improve flexibility but as Szyperski noted, there is a conflict between small components and flexibility on the one hand and large components and usability on the other ("maximizing reusability minimizes usability") [5].
- *Encapsulation.* The main motivation behind software components however, is to achieve the same as has been achieved in electronics (i.e. pluggable components). In order to achieve this, there must be a clear separation between the externally visible behavior of a component and its internal implementation. The latter must be encapsulated by the component. This feature is also referred to as information hiding or black-box behavior and is generally considered to be an important feature of the Object Oriented (OO) paradigm.
- *Composition mechanisms.* A component is used by connecting it to other components and thus creating a system based on multiple components. Components can be plugged together in many ways. These range from something as simple as a method call to more complex mechanisms such as pipes and filters or event mechanisms. Currently, there are three dominating component models (COM, CORBA and JavaBeans) these providing a

general architecture for plumbing components. Both allow for method calls (synchronous calls) and event mechanisms (asynchronous calls).

The concept of roles is based on the notion that components may have several different uses in a system. These different uses of a component in a system are dependent on the different roles a component can play in the various component collaborations in the system. The idea of role-based components is that the public interface is separated into smaller interfaces, which model these different roles. Users of a component can communicate with the component through the smaller role interfaces instead of through the full interface. The main advantage of this is that by limiting the communication between two components by providing a smaller interface, the communication becomes more specific. Because of the smaller interfaces the communication also becomes easier to generalize (i.e. to apply to a wider range of components). These two properties make the components both more versatile and reusable.

This is particularly important when component collaborations (i.e. archetypal behavior of interacting components) are to be modeled. Traditionally, the full interface of a component is considered when modeling component collaborations. Because of this, the conceptual generality of a collaboration is lost in the design as the lowest level at which modeling can be performed is the class-interface. This means that collaborations are always defined in terms of classes and the only way for components to share an interface (i.e. to be part of the same collaboration) is to inherit from a common base class. In some cases, multiple inheritance can be applied to inherit from multiple base classes, but this is generally considered to be a bad practice. By introducing roles, these collaborations can be modeled at a much more finely grained level.

If, for example, we analyze a simple GUI button, we observe that it has several capabilities: it can be drawn on the screen, it has dimensions (height, width), it produces an event when clicked, it has a label, it may display a tool tip, etc. If we next analyze a text label we find that it shares most of its capabilities with the button but sends no event when it is clicked.

The OO approach to modeling these components would be to define a common base class exposing an interface, which accommodates the common capabilities. This approach has severe limitations when modeling collaborations because in a particular collaboration, usually only one particular capability of a component is of interest. A button for instance could be used to trigger some operation in another component (i.e. the operation is executed when a user clicks the button). When using OO techniques, this must be modeled at the class-level. Even though only the event-producing capability of the button is relevant in this particular collaboration, all the other capabilities are also involved because the button is referred to as a whole.

Roles radically change this since roles make it possible to involve only the relevant part of the interface. The button in the example above, for instance, could implement a role named Event-Source. In the collaboration, a component with the role EventSource would be connected to another component implementing the role EventTarget. Similarly, the ability to display text could be captured in a separate DisplayText interface that also applies to e.g. text labels. This way of describing the collaboration is more specific and more general, more specific because only the relevant part of the interface is involved and more general because any component which supports that role can be part of the collaboration.

This idea has been incorporated into the OORam [6] method, which is discussed later in this chapter. The term *role model* will be used to refer to a collaboration of components implementing specific roles. Role models can be composed and extended to create new role models and role models can be mapped to component designs. It should be noted that multiple roles could be mapped to a single component, even if these roles are part of one role model. In the example given above, a button component could be both an EventSource and an EventTarget. This means that it is possible to model a component collaborating with itself. Of course this is not particularly useful in this example but it does show the expressiveness of roles as opposed to full class-interfaces.

From the above it can be concluded that there is no need to place many constraints on the component aspects discussed earlier, in discussing role-based component engineering. Role-based components can support multiple, typically small interfaces. The size of the component is not significant. Since multiple, functionally different components will support the same role interface, it is not desirable to take the implementation of a component into account when addressing it through one of its role interfaces.

The relation between a role and a component, which supports that role, should be seen as an "is-a" relation. The relations between roles can be both "is-a" and "has-a" relations. While hybrid components are possible (components that are only partly role-oriented), it is, in principle, not necessary to have component-to-component relations in the source code. Typically, references to other components will be typed using the role interfaces rather than component classes.

The aim of this chapter is to study role-based component engineering from a several different perspectives. In the following sections we will first motivate the use of roles by using existing metrics for object-oriented systems. Several techniques which make it possible to use roles in both design and implementation are then discussed and finally, the use of roles in object-oriented frameworks. .

Motivating the use of roles

In this section we will argue that using roles as outlined in the introduction improves an OO system in such a way that some metrics, typically used to assess the software quality of an OO system, will improve.

Kemerer & Chidamber [7] describe a metric suite for object-oriented systems. The suite consists of six different types of metrics which together make it possible to perform measurements on OO systems. The metrics are based on so-called viewpoints, gained by interviewing a number of expert designers. On the basis of these viewpoints, Kemerer and Chidamber presented the following definition of good design: *"good software design practice calls for minimizing coupling and maximizing cohesiveness"*.

Cohesiveness is defined in terms of method similarity. Two methods are similar if the union of the sets of class variables they use is substantial. A class with a high degree of method similarity is considered to be highly cohesive. A class with a high degree of cohesiveness has methods which mostly operate on the same properties in that class. A class with a low degree of cohesiveness has methods which operate on distinct sets, i.e. there are different, more or less independent sets of functionality in that class.

Coupling between two classes is defined as follows: *"Any evidence of a method of one object using methods or instance variables of another object constitutes coupling"* [7]. A design with a high degree of coupling is more complex than a design with a low degree of coupling. Based on this notion, Lieberherr et al. created the law of Demeter [8] which states that the sending of messages should be limited to

- Argument classes (i.e. any class which is passed as an argument to the method that performs the call or self),
- Instance variables.

Applied to role-based component engineering, this rule becomes even stricter: the sending of messages should be limited to argument roles and instance variables (also typed using roles).

The use of roles makes it possible to have multiple views of one class. These role perspectives are more cohesive than the total class-interface since they are limited to a subset of the class-interface. The correct use of roles ensures that object references are typed using the roles rather than the classes. This means that connections between the classes are more specific and more general at the same time. More specific, because they have a smaller interface; and more general, because the notion of a role is more abstract than the notion of a class. While roles do nothing to reduce the number of relations between classes, it is now possible to group the relations in interactions between different roles which makes them more manageable.

Based on these notions of coupling and cohesiveness, Kemerer and Chidamber created six metrics [7]:

- *WMC:* weighted methods per class. This metric reflects the notion that a complex class (i.e. a class with many methods and properties) has a larger influence on its subclasses than a small class. The potential reuse of a class with a high WMC is limited however, as such a class is application-specific and will typically need considerable adaptation. A high WMC also has consequences with respect to the time and resources needed to develop and maintain a class.
- *DIT:* depth of inheritance tree. This metric reflects the notion that a deep inheritance hierarchy constitutes a more complex design. Classes deep in the hierarchy will inherit

and override much behavior from classes higher in the hierarchy, which makes it difficult to predict their behavior.

- *NOC:* number of children. This metric reflects the notion that classes with many subclasses are important classes in a design. While many subclasses indicate that much code is reused through inheritance, it may also be an indicator of lack of cohesiveness in such a class.
- *CBO:* coupling between object classes. This reflects that excessive coupling inhibits reuse and that limiting the number of relations between classes helps to increase their reuse potential.
- *RFC:* response for a class. This metric measures the number of methods, which can be executed in response to a message. The larger this number, the more complex the class. In a class hierarchy, the lower classes have a higher RFC than higher classes since they can also respond to calls to inherited methods. A higher average RFC for a system indicates that implementation of methods is scattered throughout the class hierarchy.
- *LCOM:* lack of cohesiveness in methods. This metric reflects the notion that non-cohesive classes should probably be separated into two classes (to promote encapsulation) and that classes with a low degree of cohesiveness are more complex.

The most important effect of introducing roles into a system is that relations between components are no longer expressed in terms of classes but in terms of roles. The effect of this transformation can be evaluated by studying its effects on the different metrics:

- *WMC:* weighted methods per class. Roles model only a small part of a class interface. The amount of WMC of a role is typically less than that of a class. Components are accessed using the role interfaces. A smaller part of the interface must be understood than when the same component is addressed using its full interface.

- *DIT:* depth of inheritance tree. The DIT value will increase since inheritance is the mechanism for imposing roles on a component. It should be noted however that roles only define the interface, not the implementation. Thus while the DIT increases, the distribution of implementation throughout the inheritance hierarchy is not affected.

NOC: number of children. Since role interfaces are typically located at the top of the hierarchy, the NOC metric will typically be high. In a conventional class hierarchy, a high NOC for a class expresses that that class is important in the hierarchy (and probably has a low cohesiveness value). Similarly, roles with a high NOC are important and have a high cohesiveness value.

CBO: coupling between object classes. The CBO metric will decrease since implementation dependencies can be avoided by only referring to role interfaces rather than by using classes as types.

- *RFC:* response for a class. Since roles do not provide any implementation, the RFC value will not increase in implementation classes. It may even decrease because class inheritance will be necessary to inherit implementation only, interfaces no longer.

LCOM: lack of cohesiveness in methods. Roles typically are very cohesive in the sense that the methods for a particular role are closely related and roles will thus, typically, have a lower LCOM value.

Based on the analysis of these six metrics it is safe to conclude that:

- Roles reduce complexity (improvement in CBO, RFC and LCOM metrics) in the lower half of the inheritance hierarchy since inter-component relations are moved to a more abstract level. This is convenient because this is generally the part of the system where most of the implementation resides.

- Roles increase complexity in the upper half of the inheritance hierarchy (Higher DIT and NOC values). This is also advantageous as it is now possible to express design concepts on a higher, more abstract level that were previously hard-coded in the lower layers of the inheritance hierarchy.

Role technology

The use of roles during both design and implementation is discussed in this section. Several modeling techniques and the use of roles in two common OO languages (Java and C++) are studied.

Using roles at the design level

Though roles provide a powerful means of modeling component collaborations, the common modeling languages (e.g. UML [9] and OMT) do not treat them as first class entities. Fowler [10] suggests the use of the UML refinement relation to model interfaces. While this technique is suitable for modeling simple interfaces it is not very suitable for modeling more complex role models.

In a recent document on Reenskaugs homepage[11], the shortcomings of UML in representing component collaborations are discussed. Reenskaug defines collaboration as follows: *"A Collaboration describes how a number of objects work together for a common purpose. There are two aspects. The structural aspect is a description of the responsibilities of each object in the context of the overall purpose of the collaboration; and also the links that connect the objects* *into a communication whole. The dynamic aspect is a description of how stimuli flow between the objects to achieve the common purpose.”.*

It is essential that collaborations model the interaction of objects participating in the collaboration. In UML, a class diagram models the relations between classes. According to the UML 1.3 specification a class is defined as follows: *“A class is the descriptor for a set of objects with similar structure, behavior, and relationships.”*. As distinct from a class, an object in a collaboration has an identity. UML also provides the possibility of modeling object collaborations (Object Diagram) but Reenskaug argues that these are too specific to model the more general role models he uses in OORam, introduced in his book “Working with objects [6]. Using an UML object diagram, it is possible to express how a specific object interacts with another specific object. This diagram applies, however, only to those two objects.

In [11] Reenskaug proposes an extension to UML, which provides a more general way to express object collaborations without the disadvantage of being too general (class diagrams) or too specific (object diagrams). Essentially, Reenskaug uses what he calls *Classifier Roles* to denote the position an object holds in an object structure. Note that there is an important difference when modeling roles as interfaces only, as Reenskaug’s ClassifierRoles retain object identity whereas an interface has no object identity. Because of this it is possible to specify a relation between ClassifierRoles without explicitly specifying the identity of the objects, without giving up the notion of object identity completely, as in a class diagram. In principle, a single object can interact with itself and still be represented by two ClassifierRoles in the collaboration.

Reenskaug defines ClassifierRoles as follows: *“a named slot for an object participating in a specification level Collaboration. Object behavior is represented by its participation in the over-*

all behavior of the Collaboration. Object identity is preserved through this constraint: 'In an instance of a collaboration, each ClassifierRole maps onto at most one object'".

Catalysis [12] is a very extensive methodology based on UML, which offers a different approach to using roles in the design phase. Catalysis uses the concepts of frameworks to model component interactions, treating roles in a manner unlike that of OORam. It includes a notion of types and type models. A type corresponds to a role and a type model describes typical collaborations between objects of that type (i.e. the performance of a role in the collaboration). New type models can be composed from those existing. Type models can then be used to create components and frameworks. Unlike OORam's RoleClassifier, a type has no identity. It classifies a set of objects in the same way as a class but unlike a class it provides no implementation. This minor difference is the most important between the two notations apart from naming and methodology issues (both approaches include a development methodology).

UML in its default form is not sufficiently expressive to express the concepts Catalysis and OORAM use. The UML meta-model is however extensible and both Catalysis and OORam use this to Role-enable UML.

Using roles at the implementation level

After a system has been designed, it must be implemented. Implementation languages are typically on a lower abstraction level than the design. This means that in the process of translating a design to an implementation some design information is lost (e.g. constraints such as cardinalities on aggregation relations). Relations between classes in UML are commonly translated to pointers and references when a UML class diagram is implemented in, for example, C++. This information loss is inevitable but can become a problem if it becomes necessary to recover the design from the source code (for example, for maintenance).

With roles, a similar loss of information occurs. In the worst case, roles are translated into classes which means that one class contains the methods and properties of multiple roles. It is not possible to distinguish between the roles on the implementation level. Fortunately, languages such as Java and C++ can both be used to represent roles as first class entities (even if, in the case of C++, some simple tricks are required).

Native support for interfaces is provided in Java. More importantly, interface extension and multiple inheritance of interfaces is supported. Because of this, it is possible to create new interfaces by extending those existing and one class may implement more than one interface. This makes Java very suitable for supporting role-based component engineering, since it is easy to map the design level roles to implementation level interfaces.

The advantage of expressing roles in this way is that references to other classes can be typed using the interfaces. Many errors can be prevented by using type checking during compilation. In the case of Java, these types can also be used during run-time (i.e. two components that were developed separately but implement roles from a particular role model can be plugged together at runtime). The runtime environment will use the type information to permit only legal connections between components.

A problem with Java is that objects must often be cast in order to get the right role-interface to an object. A common example <u>are</u> the collection classes in Java which by default return Object references which need to be cast before they can be used. C++ does not have this problem since in C++, templates can often be used to address this. A similar solution in the form of a Java language extension is currently planned in an upcoming version of SUN's JDK [13].

C++ has no language construct for interfaces. Typically, the interface of a class is defined in a header file. A header file consists of a preprocessor and declarations. The contents are typically mixed with the source code at compile time. This means that the implied “is-a” relation is not enforced at compile time. Fortunately it is possible, as in Java, to simulate interfaces. Interfaces can be simulated by using abstract classes containing only virtual methods without implementation. Since C++ supports multiple inheritance, these abstract classes can be combined as in Java. This style of programming is often referred to as using *mixing classes*. Unfortunately the use of virtual methods (unlike Java interfaces) has a performance impact on each call to such methods, which may make this way of implementing roles less feasible in some situations.

Roles can also be mapped to IDL interfaces, which makes it possible to use multiple languages (even those not object-oriented) in one system. An important side effect of using component frameworks such as CORBA, COM or JavaBeans is that in order to write components for them, IDL interfaces must be defined and in order to use components, these IDL interfaces must be used. Adopting a role-oriented approach is therefore quite natural in such an environment.

As an example, consider the JButton class in the Swing framework commonly used for GUI applications in Java. According to the API Documentation, this class implements the following Java interfaces: Accessible, ImageObserver, ItemSelectable, MenuContainer, Serializable, SwingConstants. These interfaces can be seen as roles, which this class can play in various collaborations. The Serializable interface, for example, makes it possible to write objects of the JButton class to a binary stream. How this is done is class specific. However the object responsible for writing other objects to a binary stream can handle any object of a class implementing the Serializable interface, regardless of its implementation.

A problem is that many roles are associated with a more or less default implementation, slightly different for each class. However, imposing such default implementation on a component together with a role is difficult. Some approaches (e.g. the framelet approach discussed below) attempt to address this issue. An approach, which appears to be gaining ground currently is the aspect-oriented, programming approach suggested by Kiczalez et al. [14]. In this approach program fragments can be combined with an existing piece of software resulting in a new software system that has the program fragments included in the appropriate locations in the original program. However, these approaches have not yet evolved beyond the research state and adequate solutions for superimposing [15] behavior associated with roles on components is lacking.

Frameworks and roles

Why roles are useful and how they can be used during design and implementation is described in the above. In this section we argue that using roles together with object-oriented frameworks is useful. Object- oriented frameworks are partial designs and implementations for applications in a particular domain [16].

By using a framework, the repeated re-implementation of the same behavior is avoided and much of the complexity of the interactions between objects can be hidden by the framework. An example of this is the *Hollywood principle* ("don't call us, we'll call you") often used in frameworks. Developers write components that are called by the framework. The framework is then responsible for handling the often complex interactions whereas the component developer has only to make sure that the component can fulfill its role in the framework.

Most frameworks start out small, as a few classes and interfaces generalized from a few applications in the domain [17]. At this stage the framework is difficult to use as there is hardly any reusable code and the framework design changes frequently. Inheritance is the technique usually used to enhance such frameworks for use in an application. As the framework evolves, custom components, which permit frequent usage of the framework, are added. Instead of inheriting from abstract classes, a developer can now use predefined components, which can be composed using the aggregation mechanism.

Blackbox and white-box frameworks

The relations between different elements in a framework are shown in Figure 7.1.

Figure 7.1 Framework elements

The following elements are shown in this figure:

- *Design documents.* The design of a framework can consist of class diagrams (or other diagrams), written text or only an idea in the developer's head.
- *Role Interfaces.* Interfaces describe the external behavior of classes, Java including a language construct for this. Abstract classes can be used in C++ to emulate interfaces. The use of header files is not sufficient because these are not involved by the compiler in the type checking process (the importance of type checking when using interfaces was also argued by Pree & Koskimies[18]). Interfaces can be used to model the different roles in a system (for examples, the roles in a design pattern). A role represents a small group of interrelated method interfaces.

Abstract classes. An abstract class is an incomplete implementation of one or more interfaces. It can be used to define behavior common to a group of components implementing a group of interfaces.

Components. As noted before, the term component is a somewhat overloaded term and its definition requires care. In this chapter, the only difference between a component and a class is that the API of a component is available in the form of one or more interface constructs (e.g. Java interfaces or abstract virtual classes in C++). In the same way as classes, components may be associated with other classes. In Figure 7.1, we attempted to illustrate this by the "*are a part of"* arrow between classes and components. If these classes themselves have a fully defined API, we denote the resulting set of classes as a *component composition*. Our definition of a component is influenced by Szypersi's views on this subject [5] (see also chapter 1). However, in this definition, Szyperski considers components in general while we limit ourselves to object-oriented components. Consequently, in order to conform with this definition, an OO component can be nothing else than a single class (unit of composition) with an explicit API and certain associated classes which are used internally only.

Classes. Classes are at the lowest level in a framework. Classes differ from components only in the fact that their public API (Application Programming Interface) is not represented in the interfaces of a framework. Typically, classes are used internally by components to delegate functionality to. A framework user will not see those classes since he/she only deals with components.

The elements in figure 7.1 are connected by labeled arrows, which indicate the relations between these elements. Interfaces together with the abstract classes are usually called the white-box framework. The white-box framework is used to create concrete classes. Some of these classes are components (because they implement interfaces from the white-box framework). The components together with the collaborating classes are called the black-box framework.

The main difference between a black-box framework and a white-box framework is that in order to use a white-box framework, a developer must extend classes and implement interfaces [17]. A black-box framework, on the other hand, consists of components and classes which can be instantiated and configured by developers. The components and classes in black-box frameworks are usually instances of elements in white-box frameworks. Composition and configuration of components in a black-box framework can be supported by tools and are much easier for developers to perform than composition and configuration in a white-box framework.

A model for frameworks

The decomposition of frameworks into framework elements in the previous section permits us to specify the appearance of an ideal framework. In this section we will do so by specifying the general structure of a framework and comparing this with some existing ideas on this topic.

In [16], it is identified that multiple frameworks, covering several sub-domains of the application, are often used in the development of an application and that there are a number of problems regarding the use of multiple frameworks in an application:

- *Composition of framework control.* Frameworks are often assumed to be in control of the application. When two such frameworks are composed, there may be problems in synchronizing their functionality.
- *Composition with legacy code.* Legacy code must often be wrapped by the frameworks to avoid reimplementing existing code.
- *Framework gap.* The frameworks provided often do not cover the full application domain. In such cases, a choice must be made between extending one of the frameworks with new

functionality; creating a new framework for the desired functionality or implementing the functionality in the glue code (i.e. in an ad-hoc, non-reusable fashion).

- *Overlap of framework functionality.* The opposite problem may also occur if the frameworks provided overlap in functionality.

These problems can be avoided to some extent by following certain guidelines and by adhering to the model we present in this section.

We suggest that rather than specifying multiple frameworks, developers should instead focus on specifying a common set of roles based on the component collaborations identified in the design phase.

This set of roles can then be used to specify implementation in the form of abstract classes, components and implementation classes. Whenever possible, roles should be used rather than a custom interface. Role interfaces should be defined to be highly cohesive (i.e. the elements of the interface should be related to each other), small and general enough to satisfy all of the needs of the components, which use them (i.e. it should not be necessary to create variants of an interface with duplicated parts).

Subsequently, components should use these roles as types for any delegation_to other components and to fully encapsulate any internal classes. Not following this rule reduces the reusability of the components as this causes implementation dependencies (i.e. component A depends on a specific implementation, namely component B).

This way of developing frameworks addresses to some extent, the problems identified in [16]. Since role interfaces do not provide implementations, the problem of composition of framework control is avoided although, of course, it may affect component implementations. However, the smaller role interfaces should provide developers with the possibility of either avoiding or solving such problems.

The problem of legacy code can be addressed by specifying wrappers for legacy components, which implement interfaces from the white-box framework. Since the other components (if implemented without creating implementation dependencies) can interact with any implementation of the appropriate role interfaces, they will also be able to interact with the wrapped legacy components. Framework gap can be addressed by specifying additional role interfaces in the white-box framework. Whenever possible, existing role interfaces should be reused. Finally, the most difficult problem to address is the resolving of framework overlap. One option may be to create wrapper components, which implement roles from both interfaces, but in many cases this may only be a partial solution.

The use of roles in combination with frameworks has been suggested before. In [18] the notion of *framelets* is introduced. A framelet is a very small framework (typically no more than 10 classes) with a clearly defined interface. The general idea behind framelets is to have many, highly adaptable small entities which can be easily composed into applications. Although the concept of a framelet is an important step beyond the traditional monolithic view of a framework, we consider that the framelet concept has one important deficiency. It does not take into account the fact that there are components whose scope is larger than one framelet.

As Reenskaug showed in [6], one component may implement roles from more than one role model. A framelet can be considered as an implementation of one role model only. Rather than the Pree and Koskimies [18] view of a framelet as a component, we prefer a wider definition of a component which may involve more than one role model or framelet as in [6].

Another related technology is catalysis, which is also discussed earlier in this chapter. Catalysis strongly focuses on the precise specification of interfaces. The catalysis approach would be very suitable for implementing frameworks in the fashion we describe in this chapter. It should be noted though, that catalysis is a design level approach whereas our approach can, and should, also be applied at implementation time.

Dealing with coupling

From previous research in frameworks in our research group we have learned that a major problem in using and maintaining frameworks are the many dependencies between classes and components. More *coupling* between components means higher maintenance costs (McCabe's cyclomatic complexity [19], Law of Demeter [8]). We have already argued in the section on motivating the use of roles, that the use of role interfaces minimizes coupling and maximizes cohesiveness.

In this section we will outline a few strategies for minimizing coupling. There are several techniques which permit two classes to work together. That which they have in common is that for component X to use component Y, X will need a reference to Y. The techniques differ in the way this reference is obtained. The following techniques can be used to retrieve a reference:

1. *Y is created by X and then discarded.* This is the least flexible way of obtaining a reference. The type of the reference (i.e. a specific class) to Y is compiled into class specifying X but X cannot use a different type of Y without editing the source code of X' class.
2. *Y is a property of X.* This is a more flexible approach because the property holding a reference to Y can be changed at run-time.

3. *Y is passed to X as a parameter of some method.* This is even more flexible because the responsibility of obtaining a reference no longer lies in X' class.
4. *Y is retrieved by requesting it from a third object.* This third object can, for example, be a factory or a repository. This technique delegates the responsibility of retrieving the reference to Y to a third object.

A special case of technique number 3 is the delegated event mechanism such as that in Java [20]. Such event mechanisms are based on the Observer pattern [21]. Essentially, this mechanism is a combination of the second and the third techniques. Y is first registered as being interested in a certain event originating from X. This is done using technique 3. Y is passed to X as a parameter of one of X's methods and X stores the reference to Y in one of its properties. Later, when an event occurs, X calls Y by retrieving the previously stored reference. Components notify other components of certain events and those components respond to this notification by executing one of their methods. Consequently the event is de-coupled from the response of the receiving components. This coupling procedure is referred to as *loose coupling*.

Regardless of how the reference is obtained, there are two types of dependencies between components:

- *Implementation dependencies:* The references used in the relations between components are typed using concrete classes or abstract classes.
- *Interface dependencies:* The references used in the relations between components are typed using only interfaces. This means that in principle the component's implementation can be changed (as long as the required interfaces are preserved). It also means that any component using a component with a particular interface can use any other component

implementing that interface. This means, in combination with dynamic linking, that even future components, which implement the interface, can be used.

The disadvantage of implementation dependencies is that it is more difficult to replace the objects to which the component delegates. The new object must be of the same class or a subclass of the original object. When interface dependencies are used, the object can be replaced with any other object implementing the same interface. Interface dependencies are thus more flexible and should always be preferred to implementation dependencies.

In the model presented in this section, all components implement interfaces from role models. Consequently it is not necessary to use implementation dependencies in the implementation of these components. Using this mechanism is therefore an important step towards producing more flexible software.

Summary

Roles and frameworks are already combined in many programming environments (e.g. SUN's JavaBeans and Microsoft's COM). In this chapter we have argued why this is useful, how it can be performed during both design and implementation and how the idea of roles complements the notion of frameworks.

We first looked for a motivation for role-based component engineering in the form of a discussion of OO metrics. From this discussion we learned that these metrics generally improve when roles are used. By using roles, complexity is moved to a higher level in the inheritance hierarchy. This leads to a higher level of abstraction and makes the component relations more explicit (since roles are generally more cohesive than classes) while reducing coupling since implementation dependencies can be eliminated.

We then considered how roles could be incorporated in both design and implementation and found that UML in itself is too limited but can be extended in many ways (Catalysis and OORam) to support the role paradigm. Roles can also be supported on the implementation level. This is particularly easy in a language such as Java but can also be supported in C++ if the inconvenience of having extra virtual method calls can be accepted.

It was finally argued how roles can help to structure frameworks. By providing a common set of role models (either OORam style role models or Catalysis type models), interoperability between frameworks is improved and common framework integration problems can be addressed.

References

[1] Mcllroy M. D., "Mass produced software components", In *Proceedings of Report on a Software Engineering Conference Sponsored by the NATO Science Committee*, NATO Science Committee, 1968.

[2] Brown A. W. and Wallnau K. C., "The Current State of CBSE", In *Proceedings of Asia Pacific Software Engineering Conference, Workshop on Software Architecture and Components*, IEEE Computer Society, 1999.

[3] van Gurp J. and Bosch J., Design, Implementation and Evolution of Object Oriented Frameworks: Concepts & Guidelines, *Software Practice & Experiance*, volume 33, issue 3, 2001.

[4] Meyer B., *Eiffel: The Language* , Prentice Hall, 1992.

[5] Szyperski C., *Component Software Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[6] Reenskaug T., *Working With Objects*, Manning Publications, 1996.

[7] Chidamber S.R. and Kemerer C.F., A Metrics Suite for Object Oriented Design, *IEEE Transaction on Software Engineering*, volume 20, issue 6, 1994.

[8] Lieberherr K., Holland I., and Riel A., "Object-Oriented Programming: An Objective Sense of Style", In *Proceedings of OOPSLA Conference*, 1988.

[9] OMG, OMG Unified Modeling Language Specification, report version 1.3, June 1999, OMG, 1999.

[10] Fowler M. and Scott K., *UML Distilled - Applying the Standard Object Modelling Language*, Addison Wesley, 1997.

[11] Reenskaug, T., Reenskaug,T, UML Collaboration Semantics - A green paper, http://www.ifi.uio.no/~trygver/documents.

[12] D'Souza D. and Wills A. C., *Objects, Components and Frameworks: The Catalysis Approach*, Addison Wesley, 1998.

[13] JavaSoft, Add Generic Types To The Java Programming Language, http://jcp.org/jsr/detail/014.jsp.

[14] Kiczalez G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M., and Irwin J., "Aspect Oriented Programming", In *Proceedings of ECOOP*, 1997.

[15] Bosch J., Superimposition: A Component Adaptation Technique, *Journal of Parallel and Distributed Computing*, 1999.

[16] Bosch J. and others, Object Oriented Frameworks - Problems & Experiences, in Object-Oriented Application Frameworks, editors, Fayad M.E., Schmidt D.C., and Johnson R.E., Wiley & Sons, 1999.

[17] Roberts D. and Johnson R., *Patterns for Evolving Frameworks*, Addison Wesley, 1998.

[18] Pree W. and Koskimies K., "Rearchitecting Legacy systems - Concepts and Case study", In *Proceedings of First Working IFIP Conference on Software Architecture (WICSA '99)*, 1999.

[19] McCabe T.J., A Complexity Measure, *IEEE Transaction on Software Engineering*, volume 2, 1976.

[20] SUN, Sun Microsystems, Java Beans 1.01 Specification, http://java.sun.com/beans.

[21] Gamma E., Helm R., Johnson R., and Vlissidies J., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.