Role Model Based Framework Design and Integration

Dirk Riehle
UBS AG, Ubilab
P.O. Box, 8098 Zürich, Switzerland
++ 41 1 234 27 02, ++ 41 1 236 46 71
Dirk.Riehle@ubs.com or riehle@acm.org
http://www.ubs.com/ubilab

Thomas Gross
Departement Informatik, ETH Zürich
8092 Zürich, Switzerland
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

Today, any large object-oriented software system is built using frameworks. Yet, designing frameworks and defining their interaction with clients remains a difficult task. A primary reason is that today's dominant modeling concept, the class, is not well suited to describe the complexity of object collaborations as it emerges in framework design and integration. We use role modeling to overcome the problems and limitations of class-based modeling. Using role models, the design of a framework and its use by clients can be described succinctly and with much better separation of concerns than with classes. Using role objects, frameworks can be integrated into use-contexts that have not been foreseen by their original designers.

Keywords

Frameworks, design methods, design patterns

INTRODUCTION

Frameworks are a central concept of large-scale object-oriented software development. They promise increased productivity, shorter development times, and higher quality of applications [9, 10, 5, 6]. Many examples show that these goals can be reached, but even more examples show that they can be missed easily as well [19, 31]. Obviously, framework-based development of object-oriented systems is not yet a mature discipline. In particular, there is no coherent theory for the design of object-oriented frameworks and their integration into different use-contexts.

A primary reason for today's problems with designing and integrating frameworks is the dominant use of class-based modeling. Classes are excellent means for describing concepts and abstractions from an application domain, but they fail to adequately describe object collaboration behavior. Work on specifying and composing object collaborations, most prominently [14], has addressed these issues, but has not yet provided us with concepts and methods that specifically deal with framework design and integration. All approaches we know of ignore the intermediate framework level and directly jump from simple models to large-scale components and systems.

When designing a framework, developers must clarify which responsibilities an object has, on which use-contexts these responsibilities depend, and how the object combines the different responsibilities. They must further define the collaborative behavior of objects, which includes the different purposes of an object collaboration, and how they are composed. When defining how clients may interact with a framework, developers must clarify what a framework offers to them, and how they may make use of it. This includes not only the services offered by a framework, but also the requirements it puts on its context before it can be used.

In this paper, we analyze the problems of class and object collaboration complexity and their impact on framework design and integration. We introduce a role modeling approach to describe object-oriented designs. We then focus on how role modeling supports the design and integration of object-oriented frameworks. We discuss how role models address different aspects of frameworks and framework extensions, and how role models are used to define client interaction with frameworks.

The idea of role modeling is not new. The OOram methodology developed by Reenskaug et al. [24] presents a general approach to modeling objects and object collaborations using roles and role models. However, OOram's focus is on objects; it does not provide modeling concepts that specifically address and support framework design and integration. In this paper, we build on the role modeling foundation developed by prior research and explore framework design and integration based on role modeling.

Section 2 describes a pertinent set of framework design and integration problems that are addressed in this paper. Section 3 describes the role modeling metamodel, which forms the foundation of our approach. Section 4 and Section 5 describe framework design and integration concepts, respectively, based on the role modeling approach. Section 6 analyzes different aspects of the approach and discusses how far they help overcome the problems described in Section 2. Section 7 describes related work and Section 8 concludes the paper.

Dirk Riehle and Thomas Gross. "Role Model Based Framework Design and Integration." In *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '98). ACM Press, 1998. Page 117-133.

PROBLEMS IN FRAMEWORK DESIGN AND INTEGRATION

Framework design involves definition and description of the static and the dynamic aspects of a framework. A framework designer or a client who aims to understand the structure and dynamics of a framework (described by a set of abstract classes) must address a number of issues:

- *Class complexity.* To understand the purpose of a class, it is essential to understand how this class interacts with its clients. Every non-trivial class has a number of clients that use their instances for different purposes. Thus, a framework design methodology must provide means for describing interactive behavior of instances of a class as viewed from a particular use-context. An unstructured class interface, which hides the different ways of using the class, does not provide a way to identify its use in a specific context.
- *Object collaboration.* To understand how a framework works, one needs to understand how its objects collaborate at runtime ("no object is an island" [4]). Understanding collaboration behavior is important both for using the framework as well as extending it. Thus, a framework design methodology must provide means for describing the collaboration behavior of instances of the framework classes.
- *Separation of concerns.* Object collaborations themselves may become complex and may serve many different purposes. These different purposes should be kept separate to ease understanding and to increase reuse. A framework design methodology must therefore provide a way to specify single-purpose collaborative behavior, as well as a way to compose these collaboration specifications.
- *Reusable models and patterns.* To reuse experience and foster productivity, it is important to be able to reuse existing models and apply patterns. The description of these reusable models and patterns must blend well with the framework design mechanisms to make the gap between the pattern or reusable model and its application as small as possible.

Framework integration deals with the description of how clients make use of a framework by means of use-relationships. This is to be distinguished from framework extension, which deals with extending a framework by means of inheritance.

- *Client constraints.* Only in a trivial case may a client of a framework use it without fulfilling any requirements. More frequently, the framework imposes constraints on how it is to be used and requires specific behavior and capabilities from its clients (e.g., to observe protocols or to provide callback hooks). A framework integration methodology must allow the framework designer to express such constraints.
- *Unanticipated use-contexts.* Sometimes, a framework must be prepared for extension into unforeseen contexts, e.g., because client requirements cannot be determined in advance. The ability of a framework to be extended in such a case is crucial for its successful reuse. Thus, a framework integration methodology must provide concepts that let frameworks be used or extended even in unanticipated use-contexts.

For many of the problems listed above, individual solutions exist. Specifying object collaborations has gained much interest over the last years, and role modeling is probably the most promising approach [14, 24, 3]. Role objects are an important concept for framework integration [6, 7]. Yet, a coherent methodology that combines these approaches for framework design and integration is missing.

ROLE MODELING

We use role modeling as an enabling technology for framework design and integration. Role modeling helps address the aforementioned problems, but has not yet been applied in a methodological way to the large-scale development of object-oriented systems using frameworks. This section informally describes a metamodel for role model based software (micro) design to prepare the ground for our discussion of role model based framework (macro) design.

The presentation of the concepts in this paper focuses on the design level of software systems (interface architecture) and ignores the implementation level and its code structures (implementation and code architecture). Roles and role models are not first class programming language constructs, so that a gap between design and implementation arises that needs to be bridged by any given implementation anew. How to do so is largely independent of the design level.

Role and role type

A *role type* describes the view one object holds of another object. A role type is a type, so it can be described using an appropriate type specification mechanism. An object, which conforms to a given role type, acts according to the type specification. One also says that *the object plays a role* specified by a role type.

At any given time, an object may act according to several different role types. Thus, different clients may have different distinct views on an object. Also, different objects may provide behavior specified by a common role type.

As an example, consider a drawing editor framework, and a class Figure, which is the superclass of all classes of graphical objects derived from the framework. It defines a set of role types comprising, among others, the role types Figure, Child, and Subject. These role types define different behavioral aspects of a graphical figure object, i.e., an instance (of a subclass) of Figure.

- Role type Figure describes the regular drawing functionality of a graphical figure object (e.g., operations draw, hide).

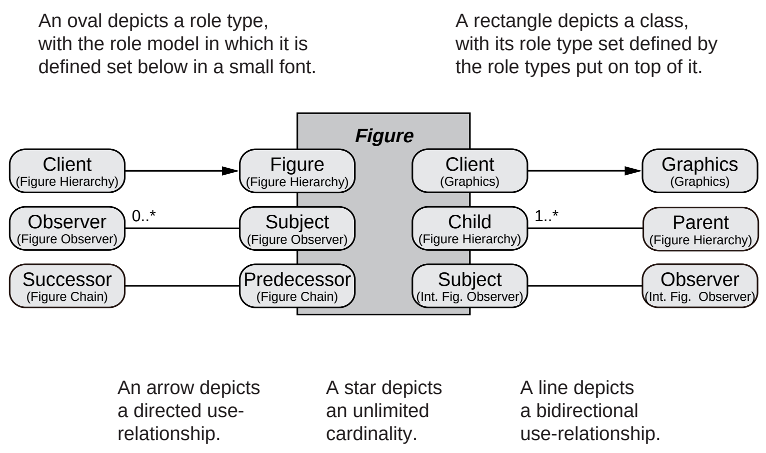

An oval depicts a role type, with the role model in which it is defined set below in a small font.
A rectangle depicts a class, with its role type set defined by the role types put on top of it.
Figure
Client (Figure Hierarchy)
Figure (Figure Hierarchy)
Client (Graphics)
Graphics (Graphics)
Observer (Figure Observer)
0..*
Subject (Figure Observer)
Child (Figure Hierarchy)
1..*
Parent (Figure Hierarchy)
Successor (Figure Chain)
Predecessor (Figure Chain)
Subject (Int. Fig. Observer)
Observer (Int. Fig. Observer)
An arrow depicts a directed use-relationship.
A star depicts an unlimited cardinality.
A line depicts a bidirectional use-relationship.

Figure 1: The class Figure and its role types.

- Role type Child describes the functionality of being a child object in an object hierarchy (e.g., operations getParent, setParent).
- Role type Subject describes the functionality of being observed for state changes by objects depending on the figure object (e.g., operations register, unregister, notify).

Figure 1 depicts the example design and introduces part of the graphical notation used in the paper.

In Figure 1, role types are qualified by a *role model* printed in a smaller font under the role type name. A role model defines a namespace for the role types that it defines, so that the role type Figure from the FigureHierarchy role model is fully qualified as FigureHierarchy.Figure, thereby distinguishing it from the class Figure. Every role type is unique. To avoid name conflicts, we qualify role types using a “RoleModel.RoleType” dot notation, e.g., FigureHierarchy.Client and Graphics.Client, when necessary.

Object and class

An *object* represents a phenomenon from an application domain, technical or non-technical. A *class* is the abstraction from several similar objects, called its instances. A class and defines a set of role types, the *role type set,* according to which its instances may play roles. The class specifies how these role types are combined in its instances. The set of role types of a class fully determines the roles an instance may play at runtime. The union of all operations defined by the role types constitutes the class interface, and the composition of all role types constitutes the type of the class.

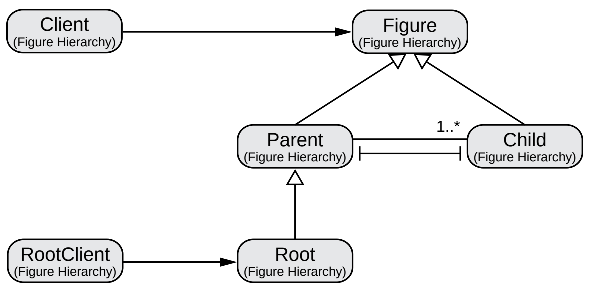
A subclass inherits the role type set of its superclass. We never had to use multiple inheritance at the design level (we use multiple inheritance in the implementation to reuse code). One reason why multiple inheritance is not used at the design level is that role types provide a satisfactory solution for those situations that might otherwise have led to the use of multiple inheritance. We have therefore restricted subclassing to single inheritance only. Standard substitutability rules can be applied, because a class specification can stand on its own, with the role types being resolved in the class interface and abstract state model.

We view role types and classes as complementary modeling concepts. The purpose of a class is to represent a domain abstraction, both its “intrinsic” properties and its behavior-oriented properties. The intrinsic properties are modeled using an abstract state space model, and the behavior-oriented properties are modeled by the role types of the class. Please note that we thereby maintain classes as modeling concepts, and do not restrict them solely to implementation.

Role model

A *role model* is the description of a (possibly infinite) set of object collaborations using role types. It focuses on a single purpose of object collaboration: a role model does not try to encompass all possible aspects of a given object collaboration. In a role model, each role type specifies the behavior of one particular object with respect to the model’s purpose. The role types relate to each other using standard object relationships like association and aggregation.

A role model is, much like a traditional class diagram, a constraining specification for a set of valid runtime object collaborations. The difference between a role model based specification and a class-based specification stems from the modeling concepts involved. With role models, objects in an object collaboration, which conforms to a role model, are known to behave as specified by the role types of the roles they are playing. However, a single role type specifies only part of an object’s overall behavior. In contrast, using class diagrams, objects in a collaboration are known to be of particular classes. Such an assertion is more restrictive, because a class already combines several different collaboration aspects (role types) and does not distinguish the different purposes of the various collaboration arrangements.



Between two role types, an arrow with a white arrowhead depicts a role-implied role constraint value.

Between two role types, a line with a block at each end (|——|) depicts a role-prohibited value.

Between two role types, an arrow with a white arrowhead at both ends depicts a role-equivalent value (not shown).

If no role constraint value is given for a pair of role types, the default role-dontcare value is assumed.

Figure 2: The Figure Hierarchy role model.

Figure 2 shows the Figure Hierarchy role model, which describes how figure objects play roles to maintain an object hierarchy (object tree). An object playing a role specified by role type Parent may maintain several objects playing a role specified by role type Child. Or, somewhat shorter (though less precise): An object playing the Parent role may maintain several objects, each playing the Child role. If the context is clear, it can expressed even more succinctly: A Parent object maintains several Child objects.

Both objects playing the Parent or Child roles are also objects playing the Figure role (this does not indicate an inheritance relationship but a role constraint, see below). An object playing the Root role also always the Parent role. A Root object represents the root of the object hierarchy and provides special management functionality.

Next to the usual object relationship descriptions, role types also relate directly to each other to constrain how objects may play several roles at once within the scope of a given role model. E.g., a role may require another role, or a role may prohibit another role. They are most conveniently expressed as a role relationship matrix [27].

Let R be the set of all role types from a given role model. Then, for every pair (A, B) of role types A and B from R, exactly one role constraint value is defined. A role constraint value is one of the following: *role-dontcare, role-implied, role-equivalent,* or *role-prohibited.*

- For a role type pair (A, B), a *role-dontcare* value states that there are no constraints on an object playing any of the roles defined by the role types A and B.
- For a role type pair (A, B), a *role-implied* value states that an object playing a role defined by role type A must always be able to play a role defined by role type B (but not necessarily the other way).
- For a role type pair (A, B), a *role-equivalent* value states that the role-implied constraint holds for (A, B) as well as for (B, A). (I.e., both roles are always available together.)
- For a role type pair (A, B), a *role-prohibited* value states that an object playing a role defined by role type A never plays a role defined by role type B and vice versa.

Role constraints are constraints on object collaborations. *Role constraints are always scoped by the role model, for which they have been defined.* Role constraints have consequences on how role types are statically assigned to classes (see next subsection).

These role constraints, in particular the role-implied constraint, should not be confused with the inheritance concept, which is a relationship type that applies to classes only. The white-headed arrows in Figure 2 represent role-implied constraints, not class inheritance relationships. While the role-implied constraint suggests equivalence with the class inheritance concept, the analogy immediately breaks down with the role-equivalent value, for which no similar class relationship type exists, as inheritance is typically considered to be non-circular.

Role models can be composed. A role model composition is a role model in which the individual role models appear unchanged, but whose role types relate to each other using any of the aforementioned constraints. The role constraints specify constraints on how roles defined by role types from the different role models come together in the composition. They do not specify the actual type compositions, which are only carried out in the context of classes and class models (see below).

Figure 3 depicts the example of a role model composition where the Figure Observer role model is composed with the Figure Hierarchy role model of Figure 2. The connection between the two role models is established by the role-equivalent constraint between the Figure role type of the Figure Hierarchy role model (i.e., FigureHierarchy.Figure)

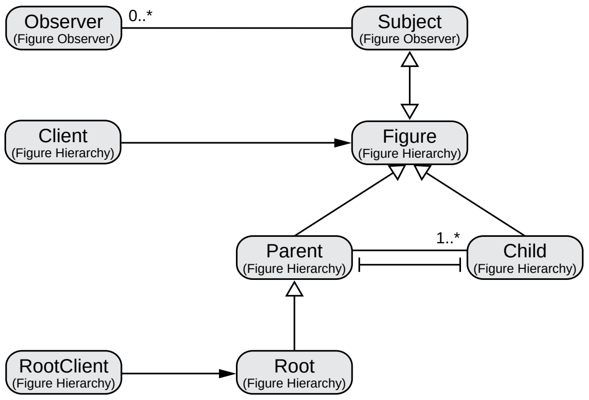


Figure 3: A composed role model.

and the Subject role type of the Figure Observer role model (i.e., FigureObserver.Subject).

Role models provide excellent separation of concerns due to their focus on one particular collaboration purpose, while traditional class diagrams necessarily intertwine all different object collaboration aspects. When composing role models, several aspects of object collaborations can be specified without prematurely committing to a class structure that might turn out to be too rigid later on.

Class model

A *class model* is a set of classes that relate to each other with any one of the following ways. First, classes may relate to each other using class inheritance. Second, classes may relate to each other by descriptions of the object relationships defined by role types in their role type sets.

Classes combine role types, and class models combine role models. Classes combine several role types from different role models. Within a class model, some (or all) of the role types of the involved role models are assigned to the role type sets of the classes. Thereby, the role models (and their role constraints) are resolved in the class model.

Role types in a role model may be left dangling for further composition, but of each role model, at least one role type needs to be in the role type set of a class. Otherwise it would not be connected with the class model.

Figure 4 shows a (partial) class model that might be used in the design of a graphical editor for UML-based object-oriented designs. The graphical editor application provides abstractions for Figure objects, which are maintained in an object hierarchy. Next to three general classes (Figure, CompositeFigure, RootFigure), the editor class model contains two classes (RectangleFigure, ClassFigure) considered specific to the UML editor. The class model can be extended with further classes like TextFigure and AssociationFigure.

The design has many aspects. Several role models express different aspects of runtime object collaborations. The class model shows how the role types of the role models and their constraints have been resolved so that the role constraints are maintained.

The Figure Hierarchy role model describes how figure objects are maintained in an object hierarchy so that composite figures may contain figures, which in turn may be composite figures that contain further figures, etc. The Figure Chain role model describes how objects forward client requests up the hierarchy, until a request is handled. The Figure Observer role model describes how clients register interest in state changes of a given figure object and are notified about them. The Internal Figure Observer (abbreviated as Int. Fig. Observer) role model describes how a Parent object observes its Child objects to react to state changes that might affect its own state.

The class model resolves the static aspects of the role constraints of the role model compositions. E.g., Figure 3 defines a role-equivalent constraint between the role types FigureHierarchy.Figure and FigureObserver.Subject. Thus, class Figure defines both role types as elements of its role type set, because the two role types must always be available together. A role-equivalent constraint between two role types A and B always requires that a class maintains both role types in its role type set, or that one of the role types is always provided by every concrete subclass of a class that defines the other role type. Another example of a role-equivalent constraint is (FigureHierarchy.Parent, IntFigObserver.Observer), which we haven't defined as an explicit role model composition, but which is directly resolved in the class model of Figure 4.

Role-implied constraints provide more freedom in assigning role types to classes by using class inheritance. Both FigureHierarchy.Parent and FigureHierarchy.Child imply FigureHierarchy.Figure, but these implications have been resolved differently. Child and Figure are provided by the same class Figure, while Parent is only introduced in the subclass CompositeFigure. Assigning Child and Figure to the same class in a class model is a stricter specification than required by the original role model. It is a convenient one, because it makes the class model simpler than it would have been had every role-implied constraint lead to a sub-

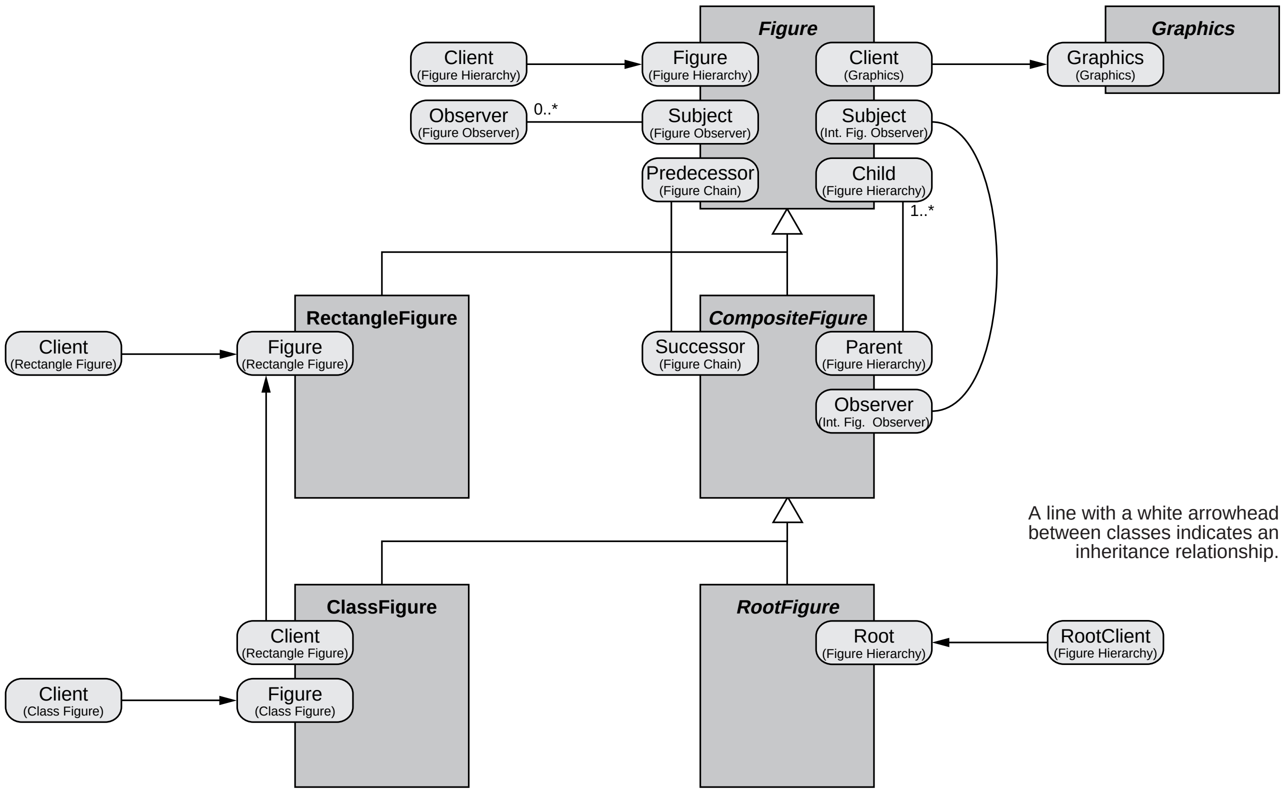


Figure
Graphics
Client (Figure Hierarchy)
Figure (Figure Hierarchy)
Client (Graphics)
Graphics (Graphics)
Observer (Figure Observer)
0..*
Subject (Figure Observer)
Subject (Int. Fig. Observer)
Predecessor (Figure Chain)
Child (Figure Hierarchy)
1..*
RectangleFigure
CompositeFigure
Client (Rectangle Figure)
Figure (Rectangle Figure)
Successor (Figure Chain)
Parent (Figure Hierarchy)
Observer (Int. Fig. Observer)
A line with a white arrowhead between classes indicates an inheritance relationship.
ClassFigure
RootFigure
Client (Rectangle Figure)
Root (Figure Hierarchy)
RootClient (Figure Hierarchy)
Client (Class Figure)
Figure (Class Figure)

Figure 4: A (partial) class model for figures for a UML editor.

class of it own. (This would have lead to many more classes and multiple inheritance diamond structures.) The consequence of this decision is that an invalid object configuration, namely a Root object playing the Child role, cannot be caught by static type checking anymore.

Role-dontcare and role-prohibited constraints provide no constraints in assigning role types to classes. Role-prohibited constraints provide no constraints, because they are scoped by their role model. As an example, consider the instantiation of the Observer pattern as, e.g., in Smalltalk's class Object (change/update mechanism). The instantiation of the Observer pattern as a role model might specify a role-prohibited constraint (Observer, Subject) so that an object might not be allowed to observe itself. However, an object may well play the Observer role in one context, and the Subject role in another context as needed, e.g., in a notification chain of observers. This is a valid object configuration, because the Observer and Subject role, while being played by the same object, are being played in different contexts, and not the same (the object is not observing itself). A role-prohibited constraint constrains the runtime object collaboration, but not the class model.

Framework

A *framework* is a class model, together with an integration role type set, and a builds-on class set. A framework covers one particular domain or a significant aspect thereof. It is a coherent unit of reuse, both by use-relationships and by extension through subclassing. (Please note that we solely focus on the design level, and ignore that frameworks typically provide reusable implementations as well.)

The *integration role type set* determines how the framework is to be used by use-relationship based clients. It contains those role types of the class model, which have not been assigned to classes, and which must be defined by client classes so that their instances can make use of objects from the framework at runtime. An element of an integration role type set is called an *integration role type.* A role model, which provides an integration role type, is called an *integration role model.*

The *builds-on class set* specifies the classes of frameworks the current framework builds on. To build on another framework, the current framework assigns some or all of the role types from the other framework's integration role type set to its classes. The builds-on class set comprises those classes of other frameworks that define role types from role models, in which integration role types are involved that are used by the current framework.

Figure 5 shows the Figure framework of the example. The integration role type set of the framework contains the role types FigureHierarchy.Client, FigureObserver.Observer, and FigureHierarchy.RootClient.

The builds-on class set contains the single class Graphics from a Graphics framework, which provides Figure objects with functionality to draw them on a device-independent graphics context. Classes of a specific framework are re-

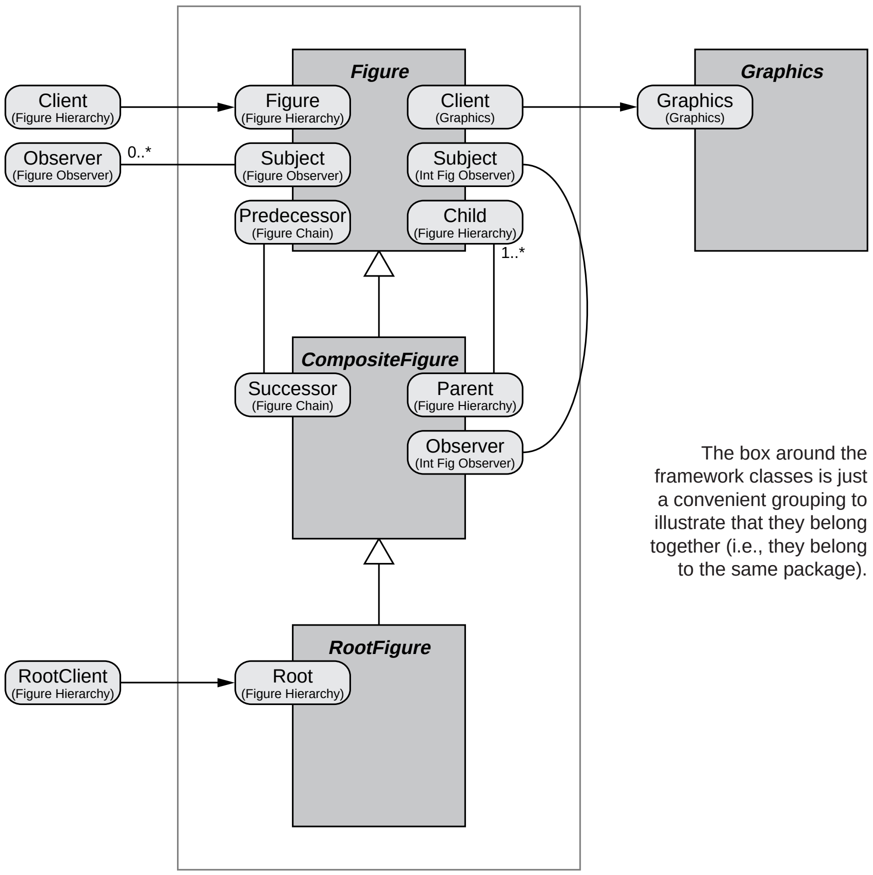


Figure
Client (Figure Hierarchy)
Figure (Figure Hierarchy)
Client (Graphics)
Graphics
Graphics (Graphics)
Observer (Figure Observer)
0..*
Subject (Figure Observer)
Subject (Int Fig Observer)
Predecessor (Figure Chain)
Child (Figure Hierarchy)
1..*
CompositeFigure
Successor (Figure Chain)
Parent (Figure Hierarchy)
Observer (Int Fig Observer)
The box around the framework classes is just a convenient grouping to illustrate that they belong together (i.e., they belong to the same package).
RootFigure
RootClient (Figure Hierarchy)
Root (Figure Hierarchy)

Figure 5: The Figure framework.

ferred to using a “Framework.Class” dot notation to qualify class names (e.g., Graphics.Graphics).

Figure 6 illustrates the Graphics framework.

The integration role type set of the Graphics framework comprises the role types Graphics.Client, Clipping.Client, Polylining.Client, Texting.Client, and Imaging.Client.

The classes Graphics.Image, Graphics.Font, and Graphics.Polygon are used by application-specific subclasses of the Figure framework and are therefore not part of the Figure framework’s builds-on class set (see section on framework design). However, they are part of the builds-on class set of the application-specific extension of the framework.

The builds-on class set of the Graphics framework is empty since it is implemented using the native API of an underlying window system.

System model

A system model is a class model. It may be the composition of an arbitrarily large set of frameworks (and framework extensions, see section on framework design).

FRAMEWORK DESIGN

We now show how the basic role modeling metamodel is used in the design of object-oriented frameworks and how it helps address the problems described in the section on problems in frameworks design and integration.

General characteristics

Frameworks have been characterized as being *black-box* or *white-box* (or both) [16]. These attributes indicate the intended usage of a framework. A black-box framework is expected to work out of the box: a client (object) can use the framework by instantiating classes and composing the instances to suit its needs. A white-box framework requires clients to supply new subclasses first, before objects can be created and composed. Many frameworks combine both characteristics by providing readily usable classes as well as abstract classes that are subclassed to provide application-specific classes.

These characteristics apply to the given framework definition of the metamodel section as well. The Figure framework is a white-box framework: Figure, CompositeFigure, and RootFigure are abstract classes that need to be subclassed for application-specific classes. Figure 4 shows such a (partial) extension for an UML diagram editor.

The Graphics framework is primarily a black-box framework that clients make direct use of without having to provide new subclasses. The object transport service frame-

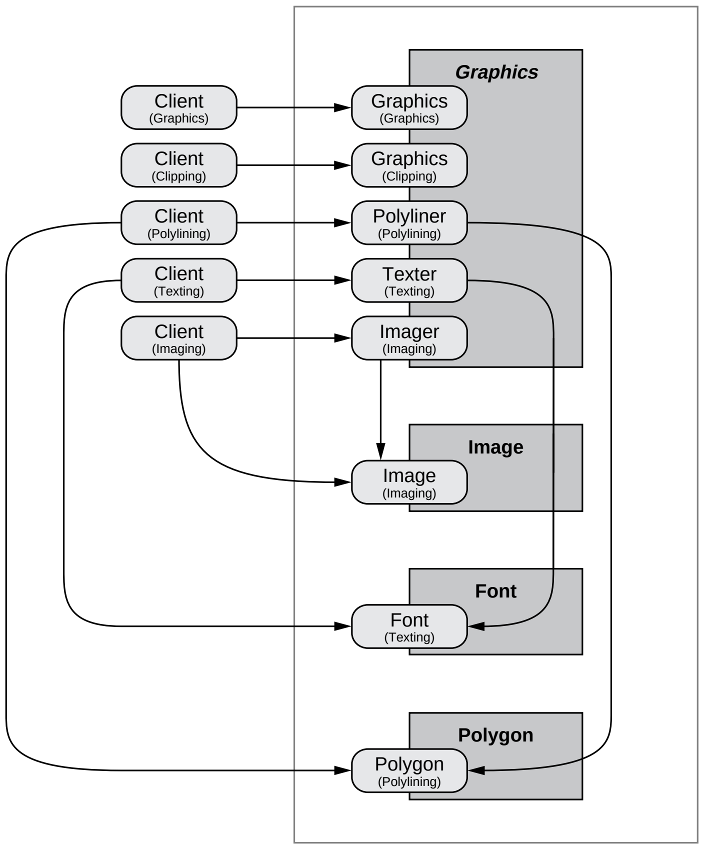


Figure 6: The Graphics framework.

work, which we described using role models [23], is a black-box framework.

Framework design

Frameworks are designed by defining and composing role models and assigning role types to classes.

We have found that design patterns are not only of occasional use when defining role models and designing frameworks, but of central importance. The application (i.e., the instantiation) of a design pattern yields a role model. We have described a catalog of object-oriented design patterns using role models [26], and illustrated their use in the design of frameworks [23].

Design patterns are ubiquitous in the design of frameworks. In [23], we discuss a large framework, which provides 12 classes and 19 role models, all of which are pattern applications. We call this effect a “high pattern density.” Almost every non-trivial role model we have seen that goes beyond a simple binary client/service role model, e.g., a ternary or an n-ary role model, can be identified as a pattern instance. Further case studies, presented in [29], support this finding with more statistical data.

When composing role models, the role constraints must be observed. Finally, the integration role type set and the builds-on class set need to be determined. The metamodel section illustrates this procedure.

Framework extension

Frameworks are extended to provide application-specific classes by subclassing. Every abstract class of the class model of a framework can be used as a superclass for an application-specific class. Extending a framework is done by providing a new subclass, which adds some role types to the role type set inherited from the superclass. A subclass that does not introduce new role types does not exist on the design-level, because it represents only a new implementation of an already known class.

Thus, a *framework extension class* is a class that is a subclass of a framework class or of another framework extension class. The set of framework extension classes of a particular application is called the *framework extension class set.*

To provide application-specific functionality, an extension class typically makes use of further frameworks. Figure 7 shows how the RectangleFigure class of the UML-editor extension of the Figure framework makes use of the Polylining role model of the Graphics framework. To do so, the RectangleFigure class defines the Polylining.Client role type for its instances. A second example is how the ClassFigure class defines the Texting.Client role type for its instances in order to make use of the Texting role model of the Graphics framework. Thus, a framework extension class

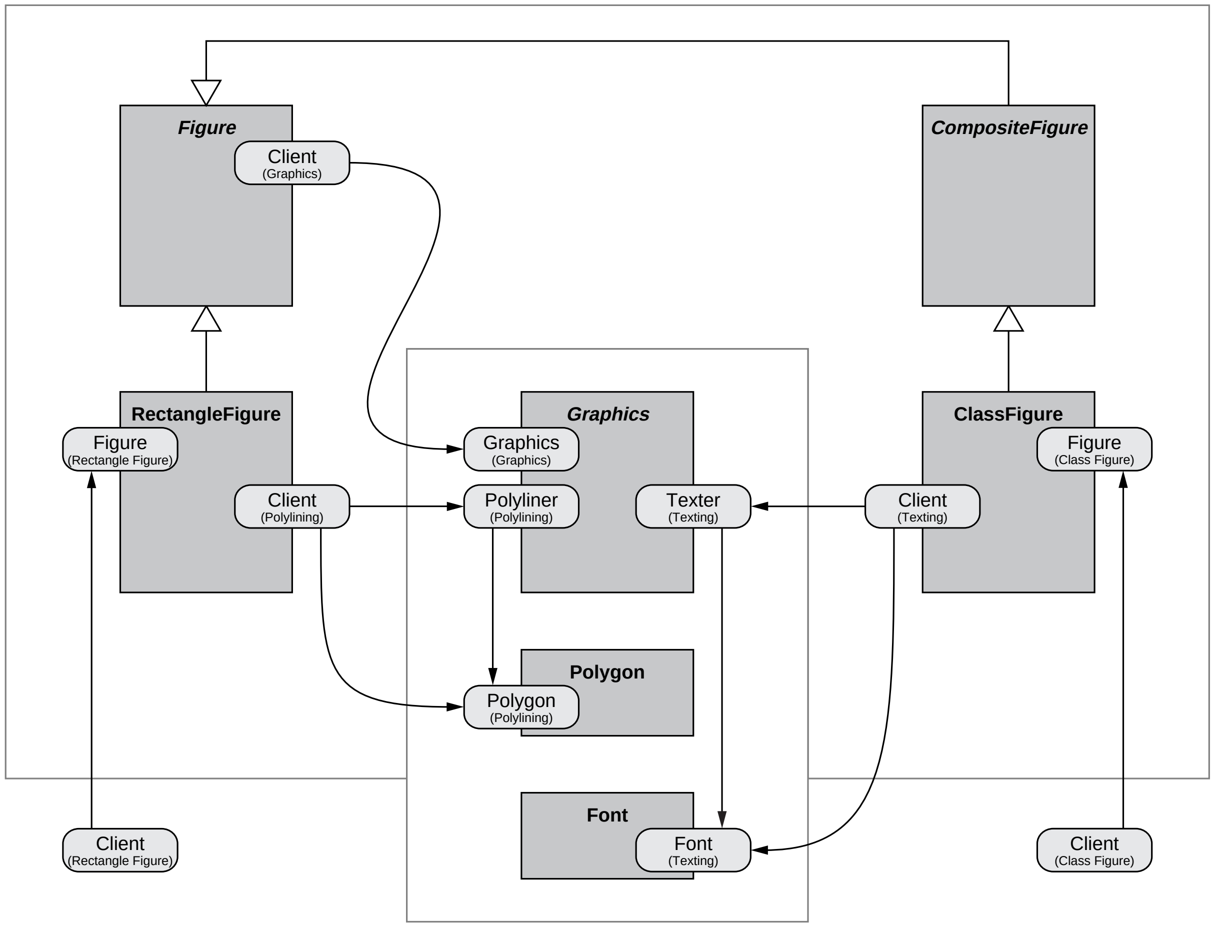


Figure 7: Coupling of Figure and RectangleFigure with Graphics.

is one of several means for integrating frameworks (see section on framework integration).

Like a framework, a framework extension defines an integration role type set, which lists the new role types it provides to clients, and a builds-on class set, which lists the classes the framework extension depends on.

FRAMEWORK INTEGRATION

This section discusses how clients build upon existing frameworks. We first discuss the most frequent case, where a framework defines its integration points using a fixed integration role type set. We then discuss how to deal with new and unforeseen clients that need to dynamically attach new roles types to a framework, based on new and unforeseen requirements.

Direct coupling

Frameworks are integrated into a larger context by that context making use of the framework. Making use of a framework means making use of classes from the framework or a given framework extension. To use a class, a client class defines for its role type set one or more role types, which stem from role models in which this class is involved in. Thus, *a role model acts as the bridge between a framework and its clients*.

Role models act as the bridge between a client and a framework, because they define the roles the client *and* the framework objects must play in order to work together properly. They define both the services offered by a framework, as well as the requirements a client has to fulfill to successfully use these services. The extent to which the requirements are made explicit depends on the chosen specification mechanism.

A client of a framework may be any class, independently of whether it stands alone as some initial application class, or whether it is a framework class or a framework extension class. It must only fulfill the constraints set up by role models of the role types from the integration role type set of the framework.

Figure 7 provides an example: the Figure and Rectangle-Figure classes of the Figure framework use the Graphics and Polygon classes of the Graphics framework. The first part of the integration is on the framework level: class Figure uses class Graphics via the Graphics role model. The second part of the integration is on the framework extension level (from the client, i.e., the Figure framework, point of view): class RectangleFigure, an extension class of the Figure framework, uses class Graphics via the Polylining role model. The integration takes place using the Graphics and

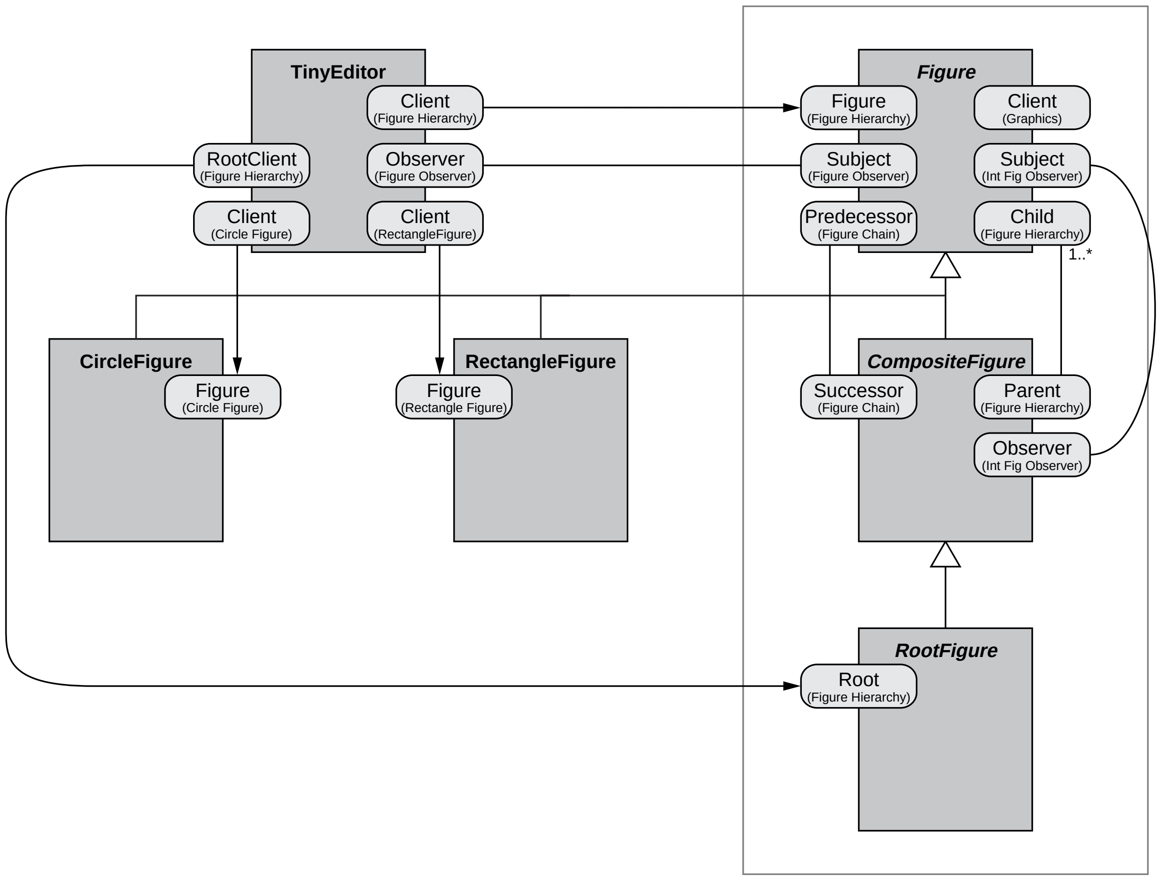

TinyEditor
Client (Figure Hierarchy)
RootClient (Figure Hierarchy)
Observer (Figure Observer)
Client (Circle Figure)
Client (RectangleFigure)
CircleFigure
Figure (Circle Figure)
RectangleFigure
Figure (Rectangle Figure)
Figure
Figure (Figure Hierarchy)
Client (Graphics)
Subject (Figure Observer)
Subject (Int Fig Observer)
Predecessor (Figure Chain)
Child (Figure Hierarchy)
1..*
CompositeFigure
Successor (Figure Chain)
Parent (Figure Hierarchy)
Observer (Int Fig Observer)
RootFigure
Root (Figure Hierarchy)

Figure 8: Integration of Figure framework for tiny editor application.

Polylining role model, respectively, and by assigning role types from these role models to the framework classes.

As a second example, consider how clients make use of the Figure framework. We consider two different types of clients, a TinyEditor client class, which takes on most of the responsibility of handling figures itself, and a UmlEditor client class, which represents a more sophisticated UML editor application.

The tiny editor application provides a TinyEditor class, which allows users to work with some basic figures, e.g., circles and rectangles. It takes on the responsibilities of creating and handling these figures, as well as placing them into a concrete implementation of a generic RootFigure instance, which acts as the containing picture object. Handling figures takes place by using the Figure Hierarchy role model as well as role models that are specific to the figure subclasses, e.g., CircleFigure and RectangleFigure. Creation and initialization of figure objects takes also place using these subclass specific role models (which is a simplifying assumption: complex initialization protocols should be role models of their own). Figure 8 shows how the TinyEditor class collects all the relevant role types in its role type set.

In contrast, a more realistic UML editor application factors the different functionality into different objects, which take on different roles. Figure 9 depicts parts of its design. The application extends the Figure framework with classes like PolygonFigure, ClassFigure, and ClassDiagram. An instance of ClassDiagram is understood here as a visually editable view on the underlying system being modeled.

We use one design aspect to illustrate the flexibility of role modeling for integrating frameworks: Class UmlEditor delegates the handling of figures (creation, moving, resizing, etc.) to Tool objects (based on the Strategy pattern [12]). For every different type of operation, there is one Tool subclass. Its instances are used to carry out the respective figure handling operation they implement. Some of these operations can be carried out using the FigureHierarchy.Figure role type, some need subclass specific role models, e.g., role model PolygonFigure for manipulation of polygons.

Integration takes place both on the framework and framework extension level. On the framework level, both classes UmlEditor and Tool, as well as specific Tool subclasses, put the FigureHierarchy.Client role into their role type set. On the framework extension level, subclasses like PolygonFigure and ClassDiagram introduce new role models. The client role types of these role models are taken on by different classes, e.g., UmlEditor, as well as specific Tool subclasses.

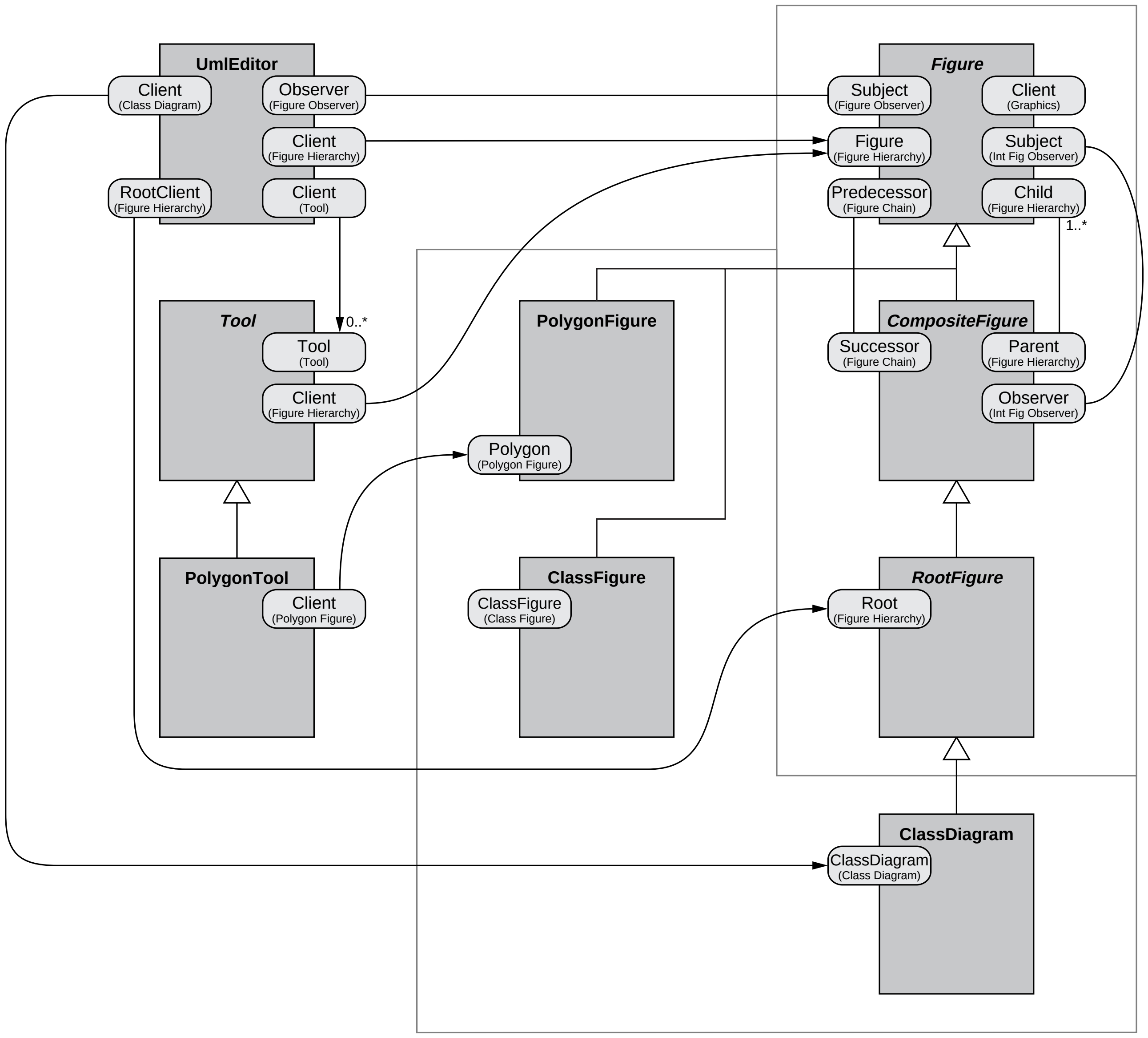

UmlEditor
Client (Class Diagram)
Observer (Figure Observer)
Client (Figure Hierarchy)
RootClient (Figure Hierarchy)
Client (Tool)
Tool
0..*
Tool (Tool)
Client (Figure Hierarchy)
PolygonTool
Client (Polygon Figure)
PolygonFigure
Polygon (Polygon Figure)
ClassFigure
ClassFigure (Class Figure)
Figure
Subject (Figure Observer)
Client (Graphics)
Figure (Figure Hierarchy)
Subject (Int Fig Observer)
Predecessor (Figure Chain)
Child (Figure Hierarchy)
1..*
CompositeFigure
Successor (Figure Chain)
Parent (Figure Hierarchy)
Observer (Int Fig Observer)
RootFigure
Root (Figure Hierarchy)
ClassDiagram
ClassDiagram (Class Diagram)

Figure 9: Integration of Figure framework for UML editor application.

We call this integration mechanism “direct coupling,” because a framework and its extensions define precisely how they may be integrated. This is done by client classes putting role types from integration role models into their role type set. This direct way of integrating frameworks is by far the most frequently used one.

Role objects

The framework integration mechanism just described is based on a statically defined set of integration role types offered by the key abstractions of a framework. Thus, a framework anticipates its use-contexts and defines a fixed set of integration points in the form of integration role models and role types. However, in large systems, new clients may require new role models for integration with a framework on which they want to build. If these role models have not been anticipated, the respective role types are not available, and integration cannot be carried out. To overcome this problem, means to dynamically attach role types to a framework’s key abstractions as required by new clients are needed.

The primary example used to illustrate this problem is the class Person and its different roles in different contexts, as found, e.g., in the banking and insurance business domain. Here, full application suites depend on specific roles of Person, which may be as diverse as Customer, Guarantor, Investor, Patient, MedicalDoctor, Employee, and Manager.

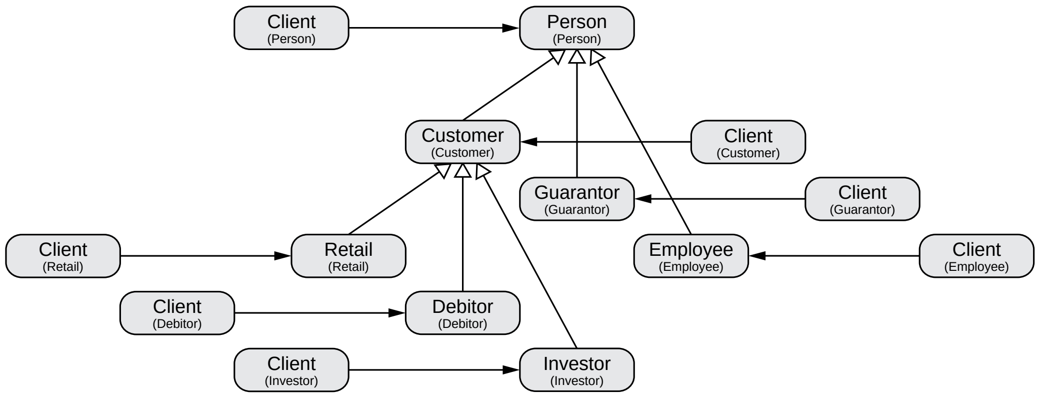


Figure 10: Composition of role models from different frameworks.

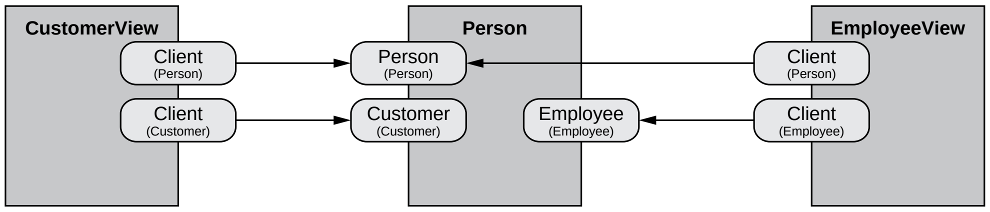


Figure 11: The Person class and two of its clients.

Figure 10 shows a set of simplified role models and their composition using role constraints. Each role model stems from a different framework. The Person and Customer role types may stem from frameworks of general interest, but the Retail, Debitor, and Investor role types come from more specific applications that typically are not available on every desktop.

In Figure 10, each role-implied constraint (A, B) expresses that an object must provide role type B if it wants to provide role type A. E.g., Customer.Customer requires Person.Person, and Investor.Investor requires Customer.Customer. (A person that plays the investor role must also always be able to play the customer role, which is its precondition, as defined in the banking domain.)

Figure 11 shows class Person with two selected role types Customer and Employee, both of which are used together with the Person role type by different clients.

The classes CustomerView and EmployeeView represent clients from a Customer Handling and Employee Handling framework, which may be used together or which may be kept separate. Each client framework introduces its specific role models, some role types of which need to be attached to the Person class of the Person framework.

If all role types are known in advance, they could be directly attached to class Person. However, this would make class Person a very heavyweight abstraction, and in a given application, only a fraction of the role types might actually be needed. Moreover, not all role models may be known in advance, and new ones might be added during system evolution when new requirements come up. Thus, one must provide means to dynamically attach role types to class Person.

In systems that do not provide adequate means to change class definitions at runtime, *role objects* can be used to attach new role types to some key abstraction. We illustrate the importance of role objects for framework integration in [6] and discuss its implementation in [7]. Here, we show how role objects fit into a general role modeling approach.

A role object is an object that represents one specific role of a core object to its clients. The role object wraps the central core object. The core object maintains its role objects for the different clients it is used by. The core object is an instance of a class, which provides the core functionality. Each role object is an instance of a class that provides one particular role type. The core object allows for the dynamic creation and deletion of role objects at configuration and runtime so that new and unforeseen role types can be attached to it on demand. The core object represents the object conglomerate and makes it appear as one logical object. It provides means for state integration, even if the logical state is spread over several physically distinct objects.

Figure 12 shows how the Person framework provides means for dynamically attaching new role types as subclasses of a predefined PersonRole class. The framework itself, providing extended role functionality, can be described well using the basic role modeling concepts.

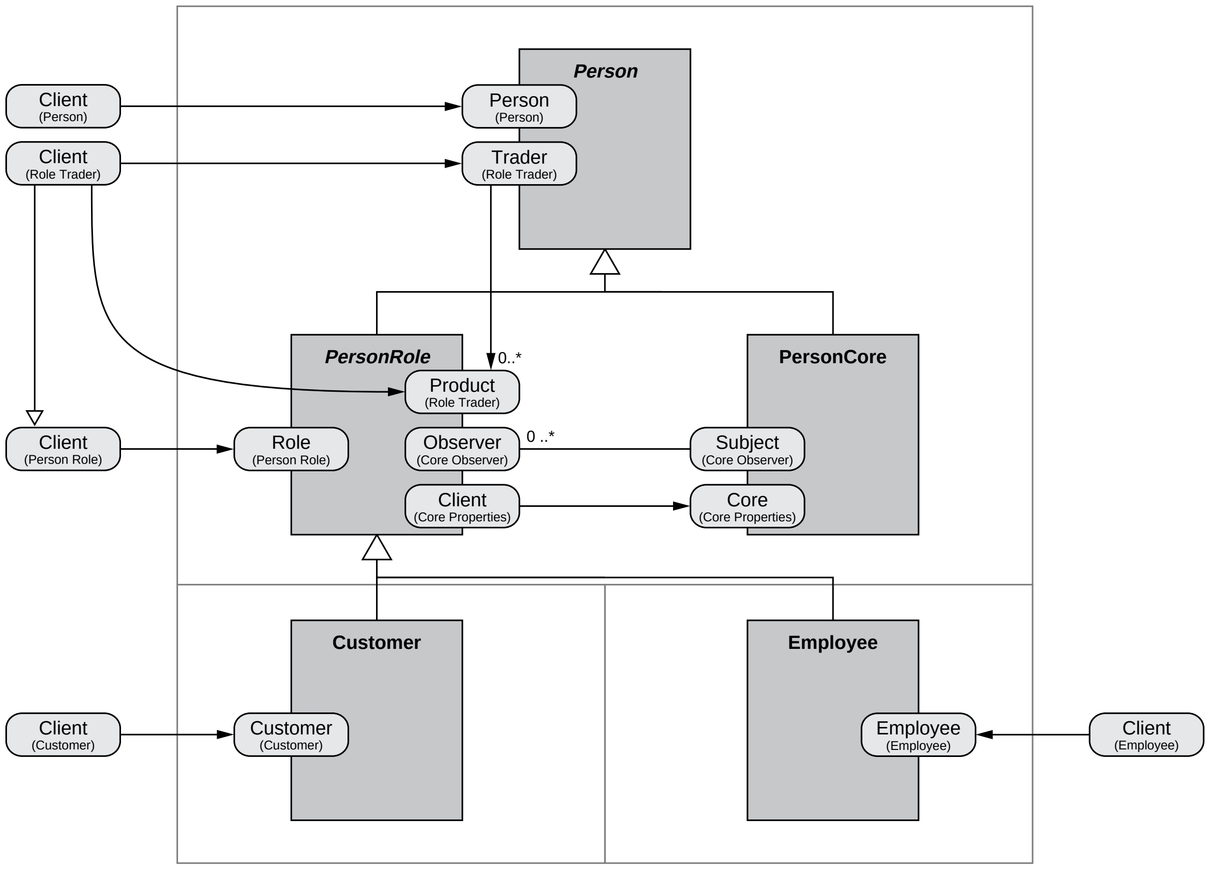


Figure 12: Part of the Person, Customer and Employee framework.

The Person class, as well as every subclass of PersonRole, provides a role type for client integration based on its primary purpose (i.e., providing the Person, Customer, or Employee role type). In addition, the Person class provides the Trader role type of the Role Trader role model, which lets clients request a specific role object from a given Person object. A role object is always an instance of a subclass of PersonRole. The PersonRole class provides the PersonRole.Role role type, which lets clients physically navigate the object structure, e.g., access the wrapped core object.

Every role object interacts with the core object to maintain a consistent overall state. In the concrete framework of Figure 12, this integration is carried out by maintaining a generically accessible list of attribute objects in the core (Property List pattern [26]), and by notifying role objects about attribute changes using the Observer pattern [12].

We make PersonRole a subclass of Person rather than an unrelated class, because it best resolves the role-implied constraints between the role types illustrated in Figure 10. For a client class of Customer, it guarantees that not only the role type Customer.Customer is available, but also Person.Person. A client object typically plays several client roles from different role models, between which it can conveniently switch if all role types are directly accessible. The implementation of a Customer role object typically directly implements the Customer role type functionality and delegates the Person role type functionality to the core object it wraps.

DISCUSSION

We have designed our role modeling metamodel in such a way that it solves the problems described in the beginning. This section discusses whether the metamodel and the concepts based on it fulfill this purpose.

Class complexity

The complexity problem of class interfaces is resolved easily by role modeling. By virtue of the role type set of a class, an explicit separation of instance collaboration concerns is achieved, which cannot be provided by a single class interface. A role type defines precisely how an instance of a class interacts in a given context (and through the role model, it defines what behavior it requires from that context to be operational).

Thus, our role modeling approach provides a sound design-level basis to programming language level concepts like Smalltalk method categories [13], Objective-C protocols [8], and Java multiple interfaces [2].

Object collaboration and separation of concerns

As noted, much of the design-level framework complexity stems from the complexity of object collaborations. Role modeling addresses both problems of properly specifying object collaborations and breaking them up into manageable pieces by means of role models. A role model describes one particular object collaboration concern. By composing role models, multiple purpose object collaborations can be described succinctly.

As already illustrated by Reenskaug [24], role modeling significantly increases intellectual manageability of object collaborations. This is achieved by the clear separation of concerns of the different object collaboration aspects as individual role models.

Reusable models and patterns

Many object-oriented design patterns can be described well using role modeling [25, 28, 23]. In particular object collaboration patterns can be described much better with role modeling than with classes. [26] contains role model based descriptions of a large number of commonly known patterns.

Reconsider the framework example of Figure 5. The Figure Hierarchy role model is an instance of the Composite pattern, the Figure Chain role model is an instance of the Chain of Responsibility pattern, the Figure Observer role model is an instance of the Observer pattern, and the Internal Figure Observer role model is another instance of the Observer pattern.

The application of a pattern leads to a (hopefully) reusable role model. As already demonstrated, role models can be reused (composed) quite naturally. In comparison to this, class hierarchies are much more difficult to compose, as Ossher and Harrison show [21]. Therefore, role modeling as described in this paper helps both to better define reusable models as well as to close the gap between design patterns and a concrete framework design.

Client constraints

By providing role models as an explicit bridge between a framework and its clients, the requirements of a framework on its context are clarified. Such an integration role model does not only specify what a framework provides, but also what it requires from its clients.

Role types are a better means for specifying these requirements than classes, because classes are too restrictive. Requiring from a client class to inherit from a framework provided class to use the framework is more restrictive than specifying that the client class must provide a specific set of role types. Using role types rather than classes as constraints on the context lets clients define their own classes without inheriting possibly unwanted and inadequate baggage.

Unanticipated use-contexts

The use of role objects is an excellent means for attaching new and unforeseen role types to classes of a framework that do not provide these role types right from the beginning. It solves the problem of a certain set of unanticipated requirements a context might pose on an already existing framework. Role objects incur a cost, though: they need to be prepared for. Thus, what must be anticipated is a particular type of unanticipated requirements, namely the need to tie in new role models into an existing framework to make it useful for clients.

Some programming languages, e.g., CLOS [22], and some programming systems, e.g., IBM Smalltalk [15], provide means to dynamically attach new functionality to classes, but this is usually not available in most mainstream programming language definitions, e.g., C++, Java, and Smalltalk. Role objects provide a means to cope with this problem on a programming language independent level.

RELATED WORK

Role models play a central role in the OOram methodology [24] and in Andersen's dissertation [3]. Both present a metamodel for conceptual modeling of object systems using roles and role models, in a style that is similar to the presentation of our metamodel summary in the metamodel section. Reenskaug and Andersen address large-scale software systems, but they only hint at frameworks as design artifacts on a level of scale between classes and large-scale components. They only discuss composed role models, with a system being viewed as the composition of a set of role models, and omit the intermediate framework level as discussed in this paper.

Our approach uses role models as a means to design and integrate frameworks. We consider frameworks to be important concepts of large-scale object-oriented system development. In practice, frameworks are an important asset of many large-scale developments, and numerous groups have recognized the value of frameworks. We thereby address an issue that has not been addressed by Reenskaug and Andersen. Frameworks are not just composed role models, but cohesive design artifacts, with well-defined integration points and ways of extending them. In addition, we have identified novel concepts like role constraints and role objects in the context of role modeling, which are concepts that help solve open problems in role model based systems modeling.

More related work on object collaboration specification and composition exists. To all, our primary critique applies: The work directly proceeds from the simple model level to the large-scale component and system level, without much consideration for an intermediate framework level.

D'Souza and Wills are working on a new UML-based methodology [32, 33, 34], which is similar to OOram in many respects, even though it does not use the concepts of role and role model. Rather, they use the concepts of interface and framework. However, they seem to use these concepts equivalently to Reenskaug's use of the role and role model concepts. Their definition of framework is different from the one we are using, and from what we know about their methodology, they do not provide an explicit approach to the design and integration on the framework level, as understood here, and as is the focus of this paper.

The work on CRC [4, 37] recognizes the importance of managing collaboration complexity. However, the approach use classes as the primary modeling concept and therefore falls short to address the problems illustrated in the section on problems in framework design and integration.

Contracts are behavioral specifications that can be composed to derive class models similar to ours [14]. Composition filters are an approach to separating object concerns [1] that also focuses on factoring behavioral aspects within a single object or component. VanHilst presents a composition methodology for collaboration based designs based on role-model alike design fragments [36].

Harrison and Ossher have defined subject-oriented programming, which deals with integrating different but related system models and implementations to build a whole system [20]. A subject is a design artifact of typically several classes and their implementation that can be composed by means of class composition operators.

The need to separate the different views on a design artifact has gained more attention through the work on viewpoints [11]. Here, different ways of specifying and using context-specific views on design artifacts are being explored.

A pertinent issue of role modeling is separations of concerns, which has received much attention recently. Most prominently, aspect-oriented programming [17] tries to describe systems as compositions of aspects. However, “aspects” are functional and behavioral aspects of objects or components and are different from role types (or role models).

Rito Silva presents a domain specific methodology for the development of concurrent systems [30]. This methodology uses class-based modeling, and with it comes all the complexity of class composition, as described by Ossher and Harrison [21].

In summary, different approaches and methodologies have addressed the problems of separation of concerns, but none of them has been extended to cover the design and integration of object-oriented frameworks on a general level. However, this area is the core theme of this paper, which reflects our observation that frameworks are a key component of the design of modern object-oriented systems.

CONCLUSIONS

Modern object-oriented systems are built from frameworks, which have been shown to be an important means of object-oriented software system construction. Yet, the design of frameworks and their use by clients to develop applications poses a number of difficult problems in terms of intellectual manageability, specification, and evolution. Many of these problems can be attributed to the sole use of classes as the primary modeling concept.

An exclusive focus on classes causes a number of problems in object-oriented framework design and integration. To overcome these problems, we develop an approach to framework design that is based on role models. Role models for framework design build on role models for object design. The use of role models for the design and integration of a framework makes clear the intent of the interaction (between objects and between the framework and its clients). Therefore role model based framework design provides designers and users of a framework with a way to deal with class and object collaboration complexity, proper modeling of separate concerns, the definition of constraints on the use-context of frameworks, and framework use after new requirements are introduced.

The design of a framework can be composed from a number of role models, each of which may be a pattern instance. Some of the role models serve as integration role models that define how clients of the framework are to use it. Effectively, these integration role models bridge the gap between a framework and its clients, and make explicit what is expected from clients that want to work with the framework. Finally, role objects show how frameworks can be integrated into unforeseen use-contexts that emerge when requirements change or are extended.

Our experience with this approach in several real-world case studies indicates that the presented concepts significantly help with framework design and integration. Role model based framework design provides advantages in addressing the core areas of framework design: managing the collaboration complexity inherent in non-trivial frameworks, description of how a framework is to be used (by clients), and devising a structure that allows a framework to work in unanticipated use-contexts. Given the importance of frameworks for the efficient construction of large-scale applications, we expect that role model based design will play an important role in the construction of future frameworks.

ACKNOWLEDGEMENTS

We would like to thank Hans Wegener and the anonymous reviewers for their comments on the paper.

REFERENCES

1. Lodewijk Bergmans and Sinan Vural. “An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach.” In *Proceedings of the 1992 European Conference on Object-Oriented Programming* (ECOOP ’92). Springer-Verlag, 1992. Page 372-395.

2. Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

3. Egil P. Andersen. *Conceptual Modeling of Objects.* Ph.D. Thesis. Oslo, Norway: University of Oslo, 1997.

4. Kent Beck and Ward Cunningham. “A Laboratory for Teaching Object-Oriented Thinking.” In *Proceedings of the 1989 Conference on Object-Oriented Programming*

Systems, Languages, and Applications (OOPSLA '89). ACM Press, 1989. Page 1-6.

5. William Berg, Marshall Cline, and Mike Girou. "Lessons Learned from the OS/400 OO Project." *Communications of the ACM* 38, 10 (October 1995): 54-64.
6. Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, and Heinz Züllighoven. "Framework Development for Large Systems." *Communications of the ACM* 40, 10 (October 1997). Page 52-59.
7. Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. "Role Object." In *Proceedings of the 1997 Conference on Pattern Languages of Programming* (PLoP '97). Washington University Department of Computer Science, Technical Report WUCS-97-34, 1997. Paper 2.1, 10 pages
8. Brad J. Cox. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1987.
9. Sean Cotter, with Mike Potel. *Inside Taligent Technology.* Addison-Wesley, 1995.
10. Mohamed E. Fayad and Wei-Tek Tsai (editors). Special Issue on Object-Oriented Experiences. *Communications of the ACM* 38, 10 (October 1995).
11. Anthony Finkelstein. "Relating ViewPoints: A Preface to Viewpoints '96." In *Proceedings of the 1996 International Workshop on Multiple Perspectives in Software Development.* ACM Press, 1996. Page 157.
12. Erich Gamma. Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
13. Adele Goldberg and David Robson. *Smalltalk-80—The Language.* Addison-Wesley, 1989.
14. Richard Helm, Ian M. Holland and Dipayan Gangopadhyay. "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems." In *Proceedings of the 1990 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '90). ACM Press, 1990. Page 169-180.
15. International Business Machines Corporation. *IBM Smalltalk User's Guide, Version 3, Release 0.* International Business Machines Corporation, 1995.
16. Ralph E. Johnson and Brian Foote. "Designing Reusable Classes." *Journal of Object-Oriented Programming* 1, 2 (June/July 1988). Page 22-35.
17. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming." In *Proceedings of the 1997 European Conference on Object-Oriented Programming* (ECOOP '97). Springer Verlag, 1997. Page 220-242.
18. Karl J. Lieberherr. *Adaptive Object-Oriented Software.* Boston, MA: PWS Publishing Company, 1995.
19. Simon Moser and Oscar Nierstrasz. "The Effect of Object-Oriented Frameworks on Developer Productivity." *Computer* 29, 9 (September 1996): 45-51.
20. William Harrison and Harold Ossher. "Subject-Oriented Programming (A Critique of Pure Objects)." In *Proceedings of the 1993 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '93). ACM-Press, 1993. Page 411-428.
21. Harold Ossher and William Harrison. "Combination of Inheritance Hierarchies." In *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '92). ACM Press, 1992. Page 25-40.
22. Andreas Paepcke. *Object-Oriented Programming—The CLOS Perspective.* MIT Press, 1993.
23. Dirk Riehle, Roger Brudermann, Thomas Gross, and Kai-Uwe Mätzel. "Pattern Density and Role Modeling of an Object Transport Service." *ACM Computing Surveys* 30, 4 (December 1998). To appear.
24. Trygve Reenskaug, with Per Wold and Odd Arild Lehne. *Working with Objects.* Greenwich: Manning, 1996.
25. Dirk Riehle. "Describing and Composing Patterns Using Role Diagrams." In *Proceedings of the 1996 Ubilab Conference, Zürich.* Edited by Kai-Uwe Mätzel and Hans-Peter Frei. Konstanz, Germany: Universitätsverlag Konstanz, 1996. Page 137-152. Originally published in *Proceedings of the 1st International Conference on Object-Orientation in Russia* (WOON '96). Edited by Alexander V. Smolyaninov and Alexei S. Shestialtynov. St. Petersburg, Russia: Electrotechnical University, 1996. Page 169-178.
26. Dirk Riehle. *A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose.* Ubilab Technical Report 97.1.1. Zürich, Switzerland: Union Bank of Switzerland, 1997. Also available from www.riehle.org.
27. Dirk Riehle. "Composite Design Patterns." In *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '97). ACM Press, 1997. Page 218-228.
28. Dirk Riehle. "Bureaucracy." In *Pattern Languages of Program Design 3.* Edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998. Page 163-186.
29. Dirk Riehle. *Framework Design and Integration: A Role Model Based Approach.* Work in progress.
30. Antonio Rito Silva. "Framework, Design Patterns, and Pattern Language for Object Concurrency." In *Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA '97).

31. Steve Sparks, Kevin Benner, and Chris Faris. "Managing Object-Oriented Framework Reuse." *Computer* 29, 9 (September 1996): 52-61.

32. Desmond D'Souza. "Collaborations: Beyond Subtypes." *Journal of Object-Oriented Programming* (January 1997): 61-66.

33. Desmond D'Souza. "Types and Classes: A Language-Independent View." *Journal of Object-Oriented Programming* (March/April 1997): 10-13.

34. Desmond D'Souza. "Frameworks in Java and Catalysis." *Journal of Object-Oriented Programming* (May 1997): 59-62.

35. Rational Software Corporation et al. *UML v1.1 Semantics.* Santa Clara, CA: Rational Software Corporation, 1997.

36. Michael VanHilst and David Notkin. "Using Role Components to Implement Collaboration-Based Designs." In *Proceedings of the 1996 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '96). ACM Press, 1996. Page 359-369.

37. Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.