Roles and Classes in Object Oriented Programming

Trygve Reenskaug

University of Oslo
Department of informatics
PO box 1080 Blindern, 0316 Oslo, Norway.
http://folk.uio.no/trygver
trygver@ifi.uio.no

Published in
http://iv.tu-berlin.de/TechnBerichte/2007/2007-09.pdf , pp 54-62
ISSN: 1436-9915

Abstract. The value of a system is greater than the sum of its parts; the system organization giving the added value. In this article we show how a system description can be split into state and behaviour parts. State is described in terms of classes and associations; behaviour is described in terms of roles and connectors. This article focuses on the behaviour part and its most important contribution is the use of selection to bind roles to objects and thus to classes. This leads to a balanced paradigm for describing the state and behaviour of object systems.

Key words: object systems, collaboration, role, object, class, data model.

1 Introduction

Objects encapsulate state and behaviour. Object state is represented in the object's instance variables. Object behaviour is specified in the object's methods. The execution of a method is triggered by the object receiving a message. The binding of message to method is dynamic and depends on the implementation of the receiving object.[1]

The value of a system is greater than the sum of its parts. The parts have their own intrinsic value, and the added value stems from the system organization. The properties of a system are similar to the properties of an unattached object. System state is the accumulated state of its objects and their associations. System behaviour is triggered by messages that the system receives from its environment and is accomplished through an organized process of message interaction between its objects.

Current mainstream programming languages are class oriented; they specify sets of objects with common properties. Class based languages work well in simple cases; but they are less than ideal in complex cases where the system as a whole tends to be hidden among the details of the classes and methods. In this article, we remedy this deficiency by showing how class centred programming can be augmented by role centred pro-

1. In some cases it also depends on the nature of the sending object.

gramming where system behaviour is specified explicitly in terms of collaborations and roles.

The role is a slippery concept. Roles cannot be defined by their shape or their constitution, only by what they do in the context of a system. *The essence of object orientation is that a system of objects collaborate to achieve a common goal.* Roles are references to participating objects; each role represents the contribution these objects make towards a system goal

The work reported in this article is part of a project we call *BabyUML* . The project goal is to create a programming environment with explicit specification of system state and behaviour; this article describes the conceptual underpinnings of this environment. The project motivation and initial ideas are presented in [1]. A project status report will appear in [2].

Section 2 introduces the *collaboration* as a context for roles. Section 3 builds an intuition about the nature of roles based on analogies with common usages of the word. Section 4 defines the concepts of *role* and *class* as they are used in BabyUML. Finally, we conclude in section 5 by summing up how roles and classes give two independent perspectives on systems of interacting objects.

2 The Collaboration

A system of interacting objects is called a *collaboration*. Its primary purpose is to describe how the system works when performing a task. Only those aspects of the objects that are required for the task need to be described. Thus, details, such as the identity or precise class of the actual participating objects are suppressed.

The UML collaborations stem from the *OOram role modelling* technology. We first used it in our own development work from the early eighties; our tool was demonstrated in the Tektronix booth at the first OOPSLA in 1986. It was first mentioned in print in an article by Rebecca Wirfs-Brock [3], and later in a JOOP article [4]. A full treatment including the parts that have still not made it into UML is in [5]. Egil Andersen gives a theory of role modelling with special emphasis on system behaviour and model inheritance (synthesis) in [6]. We use the term *collaboration* in the BabyUML project rather than *role model* because we are into programming with roles, not merely modelling with them.

As a collaboration example, consider the *Observer pattern* from the *Design Patterns* book [7]. This pattern maintains consistency between related objects called `subject` and `observers`. The `subject` is an object that holds some state. The `observers` are objects that need to be informed about any change to this state.

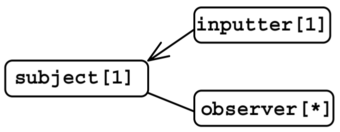


Figure 1 The Observer collaboration.

Figure 1 show the Observer pattern as a collaboration. There are three roles that represent the participating objects. The role names are aliases for the objects that will occupy the respective positions in the structure during an execution.

Figure 2 shows how the Observer collaboration maintain consistency between the `subject` and `observer` objects when the state of the `subject` object is changed by a stimulus from an `inputter` object. The diagram is a BabyUML sequence diagram that describes sequential interaction. A filled arrow denotes message transmission. A thin, vertical rectangle denotes a method execution. Any number of objects may be bound to the `observer` role; they work in lock-step so that their updates appear to occur simultaneously.

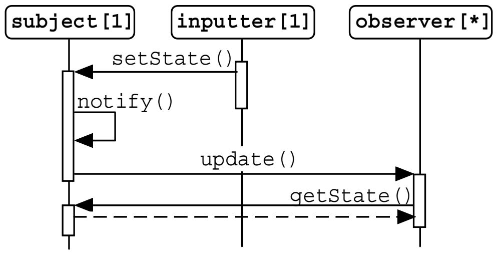


Figure 2 The Observer pattern interaction

Many objects may play the `observer` role in different contexts and at different times, but we are only concerned with the objects that play the role in an occurrence of the interaction. A role may be played by many objects and an object may play many roles. In this example, an object playing the `inputter` role often also plays the `observer` role. The object diagram in Design Patterns [7] mandated this by showing two objects called `aConcreteObserver` and `anotherConcreteObserver` respectively; the first also playing the `inputter` role.

Every message has a sender and a receiver, both are equally important from the perspective of system design. The collaboration reflects this; there are no messages without both a sender and a receiver. This in contrast with classes where the handling of received messages is specified explicitly, while the sending of messages is hidden within the methods.

The BabyUML project has UML in its name to indicate that we see UML [8] as a rich source of alternative and consistent ways of describing system behaviour; the sequence diagram of Figure 2 is a specialization of one of them.

3 The Word *Role* in Common Usage

We will now look at how the word *role* is generally used and see how this usage can help us build an intuition about our use of the word in object oriented programming.

Most people will probably associate the word *role* with a part played by an actor in a theatrical performance. Oxford [9] gives its origin: *"from obsolete French roule 'roll', referring originally to the roll of paper on which an actor's part was written."* (Note that this means the script, not the performance!) Oxford [10] has a fitting definition: *"Actor's part; one's function, what one is appointed or expected or has undertaken to do."* This maps nicely on to our use of roles for naming the participants in a system of collaborating objects; the role is a reference to an object[1]; it represents the object's function, what it is delegated or expected or is required to do.

A collaboration is analogues to a play where its code is analogues to the written drama. A role is a part in a collaboration just as a role is a part in a play. An Object plays a role in an execution of a collaboration; an actor plays a role in a particular performance of a play. An Object is selected to play a particular role at a certain time and in a certain context. An actor is selected to play a particular role in a particular performance.

An actor interprets a given role in his personal way; other actors do it differently. An object playing a particular role is selected from a pool of candidate objects. These objects may play the role differently because they can be instances of different classes even if they all satisfy the role's requirements.

Steven Pinker in *How the Mind Works* [11] claims that the meaning of a system comes from the meaning of the parts *and* from the way they are combined. Consider this very simple example taken from the Pinker's book:

> The baby ate the slug
> The slug ate the baby.

Each of the three words "baby", "ate", and "slug" has its own meaning. The two diametrically different meanings arise from the roles played by the words in the sentences. Analogically, an object system gets part of its meaning from its objects. It gets the rest of the meaning from the roles played by the objects and the way those roles are organized. As an example, consider how the Observer pattern is described in terms of roles in Figure 1 and Figure 2. The roles tell a part of the story; it is only completed when the roles are bound to actual objects.

4 Roles and classes -- a Set of Definitions

The value of a system of interacting objects stems partly from the value of its objects taken separately, and partly from the way the objects are organized to represent system state and behaviour. Figure 3 illustrates how a system can be organized to realize its behaviour. A collaboration describes how the system performs a task. Objects are represented by the roles they play as they make their contribution towards this task. These objects are selected at run time from a pool of objects by some selection mechanism. The selected objects are instances of one or more classes, thus completing the bridge from the performance of a task via roles to classes.

Figure 3 The bridge from role to class

$$\begin{tikzcd}
\textbf{collaboration} \arrow[r, "\textit{reference}"] & \textbf{role} \arrow[r, "\textit{select from}"] & \textbf{objects} \arrow[r, "\textit{instance of}"] & \textbf{classes}
\end{tikzcd}$$

1. One role can reference several objects simultaneously. They will all serve the same purpose in the Collaboration, but we can assume a single object without loss of generality.

The binding between role and object is dynamic because an object may play different roles and a role may be bound to different objects. The result of the selection can depend on many things such as the nature of the task, the process itself, and the current state of the objects. Contrast with the static binding between object and class; an object is an instance of the same class throughout its lifetime. The notions of role and class are orthogonal. A role represents the usage of an object while the class represents its construction.

We will now propose a definition for each of the parts in Figure 3. We end in section 4.5 with a discussion of the dynamic step of binding roles to objects.

4.1 The Object

The *object* is the atom of object oriented data processing systems. Objects encapsulate state and behaviour and have the following properties:

- *Identity:* An immutable property that distinguishes an object from all other objects.
- *State:* The object’s attributes and their values.
- *Interface:* The set of messages that can be received by the object.
- *Behaviour:* The methods that specify the object's processing of incoming messages. This includes changing the object state and sending messages to itself or other objects.
- An object is an instance of the same class throughout its lifetime.

4.2 The Role

The concept of *role* conforms to the common usage of the word:

- A role represents a functionality.
- This functionality can be utilized in its collaboration with other roles.
- A role is bound to one or more objects selected from a universe of objects. These selected objects are called the *players* of the role.
- A role delegates the performance of its functionality to its players.
- A role specifies requirements for its players. An important property is the set of messages that its players must understand.
- The required properties can be implemented by several classes so the players need not be instances of the same class. Different objects can thus perform the same role in different ways.

4.3 The Collaboration

A *collaboration* is a structure of interconnected roles that enables the system to perform one or more tasks.

- A collaboration describes a structure of collaborating roles, each performing a specialized function, which collectively accomplish some desired functionality.
- A collaboration structure constitutes a graph where the nodes are roles and the edges are the message interaction paths.

4.4 The Class

A class is a specification of the state and behaviour of objects. In the context of a system, an object's state is part of the system state and an object's behaviour is part of the system behaviour.

- The state is specified as a set of attributes.
- The behaviour is specified as a dictionary binding messages to methods where a method is a piece of executable code.
- The instances of a class are objects. They are all different, but they share the features described by the class.
- Classes can be organized in an inheritance hierarchy. This hierarchy is independent of the message interaction structures defined by the collaborations.

4.5 Binding roles to objects by dynamic selection

The concepts discussed above are all well known and there are well established standards for applying them to software design. UML 2.x [8] can be taken as a starting point. Roles are here defined as properties of collaborations in UML; this corresponds well with our notion of a role. UML class diagrams can be used to model the system data.

The crucial point in Figure 3 is the link between *role* and *objects*. It is annotated by *select from*; this signifies that objects are dynamically selected from a set of relevant objects to play the roles. Many different selection mechanisms can be used. We will here consider two; the first is a very simple one and the second more general.

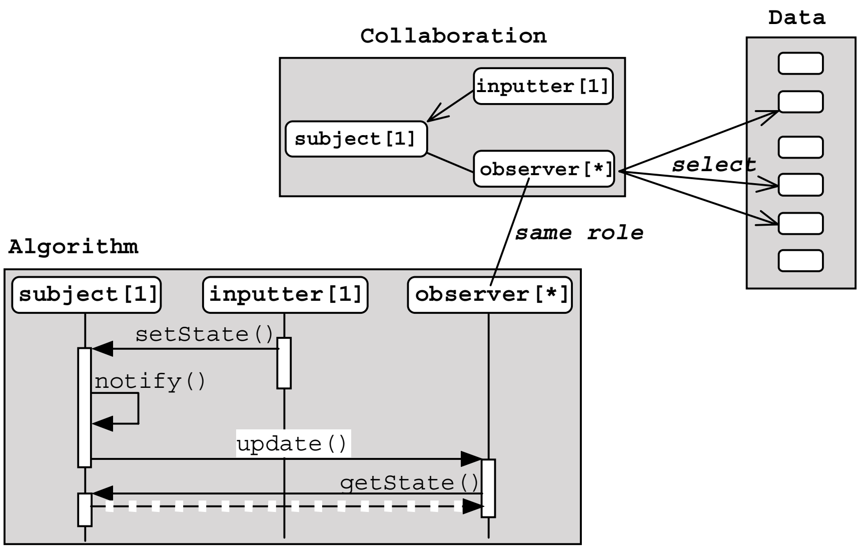
Consider the Observer pattern discussed in section 2. The pattern in [7] specifies that the `subject` has a list of `observers`, each conforming to the simple interface needed to play that role. This means that an object playing the `subject` role must understand messages such as `addObserver()` and `removeObserver()`. The selection mechanism is very simple; any object that happens to be in the list of `observers` will receive the `update()`-message.

A more elaborate example is based on the *Data-Collaboration-Algorithm (DCA)* paradigm, a paradigm that makes the collaboration a first class program element [2]. Figure 4 illustrates it with a variant of the Observer pattern:

- The *Data* part is responsible for knowing all relevant objects. In a typical application of the Observer pattern, they can be model, view, and controller objects in the MVC paradigm. (In Java, the Data part can be implemented in a class defining a "micro database", see [2].)
- The *Collaboration* part is an object that has a method for each role. These methods dynamically select the appropriate player objects. In principle, the methods should perform the selection on each call to ensure up-to-date mapping. (In [2], the nature of the example is such that the result of the query depend on the state of the objects and this state changes on every execution of the Algorithm).
- The *Algorithm* part is responsible for managing the flow of messages during an interaction. The algorithm references roles by name, and delegates to the collaboration to bind the roles to objects. In our variant of the Observer pattern, the code could look like this:

```
for (Observer obs : collab.observers()) {
     obs.update();
}
```

Figure 4 The DCA paradigm (Data-Collaboration-Algorithm)



To make the example more concrete, imagine a *Dog Breeders Association* that keeps a registry of its of pedigree dogs. There are also dog shows where dogs receive prizes according to their merits. After a show, the registry dog records shall be updated with prizes received. We let the `subject` role be played by an object that represents the show and let the `observer` role be played by the registry objects that represent the prized dogs. The code would first set

```
subject := show
```

and then select the `observers` from the dog registry (here coded in an unspecified pseudo-language):

```
define observers as
     select prized from dogregistry
     where priced is in subject.winnersList
```

The algorithm can then simply augment each `observer` with the relevant prizes from the `winnersList`.

One advantage of the Observer pattern defined in [7] is that it provides loose coupling between `subject` and `observer`. This DCA variant is even looser; the `subject` does not know the `observers` and the `observers` do not know the `subject`. The link between them only exists during the transfer.

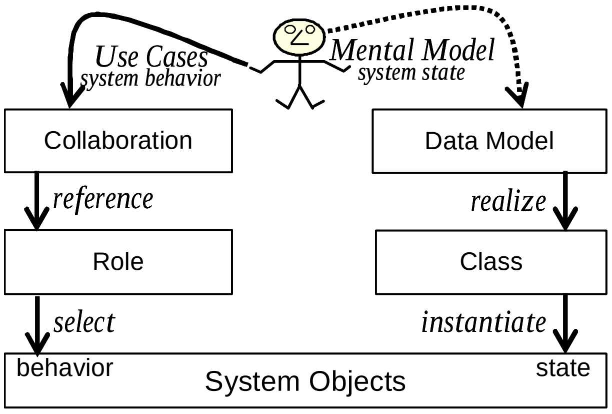
5 Conclusion

Objects have state and behaviour; they are specified separately in the instance variables and in the methods. Systems also have state and behaviour; we separate their specification in the two paths of shown in Figure 5. *System state* is specified in a *Data Model*; e.g., in the form of a UML class diagram showing the classes with their attributes and associations. The classes specify the attributes (state) of the objects; the associations are specified in the Data Model. *System behaviour* is specified in the system's collaborations. The collaboration roles specify the requirements that any object playing that role must satisfy. Finally, the methods in the participating objects are implemented in the corresponding classes so as to satisfy these requirements.

We see that the attributes of a class stem from the system's Data Model; while the methods of a class stem from the behaviour required for its instances.

The system environment is here symbolized by a human user. The way we describe systems is recursive; the environment could have been external objects and our system would then have been a subsystem.

Figure 5 Roles and classes in object oriented programming



The architecture of Figure 5 can be implemented in a language such as Java. The problem is that there is no Java language construct for explicitly specifying roles and collaborations. The consequence is that their implementation will be scattered around in the code.

The goal of the BabyUML project is to make system state and behaviour explicit by extending object oriented programming with additional projections. *Programming with Roles and Classes; the BabyUML Approach* [2] reports on the current state of the project. It includes the results of a Java experiment and the beginnings of an IDE for the specification of systems of interacting objects.

Systems are characterized by their state and behaviour. In this article, we have focused on system behaviour and have shown how it can be described in terms of collaborations and roles. We have also shown how roles can be bound dynamically to objects and this classes. The BabyUML project wants to make system behaviour explicit in the code, but our current results are fully applicable to system modelling and design.

Steven Pinker in *How the Mind Works* [11] claims that we have two independent ways of dealing with complexity; one is grouping things according to their properties, another is grouping things according to what they are used for. We use the class for the first grouping and the role for the second. Both classes and roles are essential abstractions that we need in order to master the complexity of the world round us. In this article, we have shown that the role is the atom of system behaviour. *The role should, therefore, be granted the status of an ontological primitive.*

References

[1] Reenskaug, T,: *The BabyUML discipline of programming (where A Program = Data + Communication + Algorithms).* SoSym **5**,1 (April 2006). DOI: 10.1007/s10270-006-0008-x. [web page] http://heim.ifi.uio.no/~trygver/2006/SoSyM/trygveDiscipline.pdf

[2] Reenskaug, T.: *Programming with Roles and Classes; the BabyUML Approach;* A chapter in *Computer Software Engineering Research;* ISBN: 1-60021-774-5; Nova Publishers; Hauppauge NY; 3rd Q. 2007; [WEB PAGE] http://folk.uio.no/trygver/2007/babyUML.pdf

[3] Rebecca J. Wirfs-Brock , Ralph E. Johnson: *Surveying Current Research in Object-Oriented Design.* Comm ACM **33**(9):104-124. September 1990.

[4] Reenskaug, T.; et.al. *ORASS: seamless support for the creation and maintenance of object-oriented systems;* JOOP, **5** 6 (October 1992); pp 27-41.

[5] Reenskaug et.al.: *Working with objects. The OOram Software Engineering Method;* ISBN 1-884777-10-4; Manning, Greenwich, CT. 1996; Out of print. Early version in [web page] http://heim.ifi.uio.no/~trygver/1996/book/WorkingWithObjects.pdf

[6] Andersen, E. P.; *Conceptual Modeling of Objects. A Role Modeling Approach.* D.Scient thesis, November 1997, University of Oslo. [web page] http://heim.ifi.uio.no/~trygver/1997/EgilAndersen/ConceptualModelingOO.pdf

[7] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J,; (GOF) *Design Patterns*; ISBN 0-201-63361-; Addison-Wesley, Reading, MA. 1995.

[8] Unified Modeling Language: Superstructure. Version 2.1.1; Object Management Group; document formal/2007-02-05; [web page] http://www.omg.org/cgi-bin/doc?formal/07-02-05.pdf

[9] *AskOxford.* Oxford Dictionaries.[web page] http://www.askoxford.com/?view=uk

[10] Fowler, H. W.; Fowler, F. G.; McIntosh, E.: *The Concise Oxford Dictionary of Current English.* Fifth Edition; ISBN 0 19 861107 2; Claredon Press, Oxford 1975.

[11] Pinker, S.; *How the Mind Works*; ISBN 0-393-04535-8; Norton; New York, NY, 1997.

Acknowledgements

The author is very grateful to James O. Coplien for his continued interest over many years, his valuable insights, and not least of all, our very challenging and productive discussions. The author is also grateful to an anonymous reviewer for valuable comments.