

UML 2 AND THE UNIFIED PROCESS

SECOND EDITION

PRACTICAL OBJECT-ORIENTED ANALYSIS AND DESIGN

JIM ARLOW

ILA NEUSTADT





Praise for *UML 2 and the Unified Process, Second Edition*

"OMG's UML 2 standard defines UML very systematically and thoroughly, but lacks how to actually apply UML 2 in a real-world project. That is where *UML 2 and the Unified Process, Second Edition,* comes in. The book manages to convey the practical use of UML 2 in clear and understandable terms with many examples and guidelines. Even for people not working with the Unified Process, the book is still of great use. *UML 2 and the Unified Process, Second Edition,* is a must-read for every UML 2 beginner and a helpful guide and reference for the experienced practitioner."

—Roland Leibundgut, Technical Director, Zühlke Engineering Ltd.

"The authors provide great detail on the constructs of the UML and how they support the Unified Process. This book is a good starting point for organizations and individuals who are adopting UP and need to understand how to provide visualization of the different aspects needed to satisfy it."

—Eric Naiburg
Market Manager, Desktop Products
IBM Rational Software

"Many books today cover either UML or the Unified Process (UP), but not both. Arlow and Neustadt have filled this void with a book that is a perfect blend of UML and the UP. The authors provide a wealth of experience invaluable to novice modelers and seasoned OO analysts and designers. The logical organization based on the UP workflows and the unique presentation style of using activity diagrams to provide book and chapter overviews make reading less demanding. Both practitioners and students will certainly keep this book within close reach all the time."

—Ishan De Silva
Software Engineer
Millennium Information Technologies, Sri Lanka

"If you are looking for a cookbook, read something else. This book will make you think! It explains all the syntactic elements of UML, but more importantly it gives practical advice on how and when to use—or not to use—UML. You will learn to think about the role that modeling plays in the development process. This knowledge enables you to decide for yourself how and when to use UML to gain the optimal situation for your project. *UML 2 and the Unified Process, Second Edition,* will prepare you for successfully using UML."

—Jos Warmer
Ordina System Integration & Development, The Netherlands

"The authors have crafted a book that brings two important topics, UML and the Unified Process, together in one volume. *UML 2 and the Unified Process* is an excellent reference for UML 2. It tells you what UML can do and how to apply it to the analysis and design disciplines of the Unified Process. This book belongs on every professional's bookshelf."

—Gary Pollice
Professor of Practice, Computer Science
Worcester Polytechnic Institute

8.3.3 Analysis class rules of thumb

Here are some rules of thumb for creating well-formed analysis classes.

- About three to five responsibilities per class – typically, classes should be kept as simple as possible, and this usually limits the number of responsibilities that they can support to between three and five. Our previous example of a `ShoppingBasket` is a good example of a focused class with a small and manageable number of responsibilities.
- No class stands alone – the essence of good OO analysis and design is that classes collaborate with each other to deliver benefit to users. As such, each class should be associated with a small number of other classes with which it collaborates to deliver the desired benefit. Classes may delegate some of their responsibilities to other "helper" classes that are dedicated to that specific function.
- Beware of many very small classes – it can sometimes be hard to get the balance right. If the model seems to have lots and lots of very small classes with just one or two responsibilities each, then you should look at this very carefully with a view to consolidating some of the small classes into larger ones.
- Beware of few but very large classes – the converse of the above is a model that has few classes, where many of them have a large number (> 5) of responsibilities. The strategy here is to look at these classes in turn and see if each can be decomposed into two or more smaller classes with the right number of responsibilities.
- Beware of "functoids" – a functoid is really a normal procedural function disguised as a class. Grady Booch tells the amusing anecdote of a model of a very simple system that had thousands of classes. On closer inspection, each class had exactly one operation called `doIt()`. Functoids are always a danger when analysts accustomed to the technique of top-down functional decomposition approach OO analysis and design for the first time.
- Beware of omnipotent classes – these are classes that seem to do everything. Look for classes with "system" or "controller" in their name! The strategy for dealing with this problem is to see if the responsibilities of the omnipotent class fall into cohesive subsets. If so, perhaps each of these cohesive sets of responsibilities can be factored out into a separate class. These smaller classes would then collaborate to implement the behavior offered by the original omnipotent class.
- Avoid deep inheritance trees – the essence of designing a good inheritance hierarchy is that each level of abstraction in the hierarchy should have a well-defined purpose. It is easy to add many levels that don't really

serve any useful purpose. In fact, a common mistake is to use inheritance to implement a kind of functional decomposition where each level of abstraction has only one responsibility. This is, in every respect, pointless and just leads to a complex, difficult to understand model. In analysis, inheritance is only used where there is a clear, and obvious, inheritance hierarchy arising directly from the problem domain.

In the last bullet, we need to clarify what we mean by a "deep" inheritance tree. In analysis, where the classes represent business things, "deep" would be three levels of inheritance or more. This is because business things tend to form inheritance hierarchies that are broad rather than deep.

In design, where the tree consists of classes from the solution domain, the definition of "deep" depends on the implementation language you are targeting. In Java, C++, C#, Python, and Visual Basic, we still consider three or more levels to be deep. In Smalltalk, however, inheritance trees can go much deeper than this, due to the structure of the Smalltalk system.

8.4 Finding classes

In the rest of this chapter we consider the core issue of OO analysis and design, finding the analysis classes.

As Meyer points out in *Object Oriented Software Construction* [Meyer 1], there is no simple algorithm for finding the right analysis classes. If such an algorithm did exist, then it would amount to an infallible way to design OO software and this is just as unlikely as finding an infallible way to prove mathematical theorems.

Still, there are tried and tested techniques that lead toward a good answer, and we present them here. They involve analyzing text and interviewing users and domain experts. But ultimately, despite all the techniques, finding the "right" classes depends on the perspective, skill, and experience of the individual analyst.

8.4.1 Finding classes by using noun/verb analysis

Noun/verb analysis is a very simple way of analyzing text to try to find classes, attributes, and responsibilities. In essence, nouns and noun phrases in the text indicate classes or attributes of classes, and verbs and verb phrases indicate responsibilities or operations of a class. Noun/verb analysis has been used for many years and works well as it is based on a direct analysis of the language of the problem domain. However, you have to be very aware of synonyms and homonyms as these can give rise to spurious classes.

In noun/verb analysis you analyze text. Nouns and noun phrases indicate classes or attributes. Verbs and verb phrases indicate responsibilities or operations.

You also have to be very careful if the problem domain is poorly understood and defined. In this case, try to collect as much information about the domain from as many people as possible. Look for similar problem domains outside your organization.

Perhaps the trickiest aspect of noun/verb analysis is finding the "hidden" classes. These are classes that are intrinsic to the problem domain but that might never be mentioned explicitly. For example, in a reservation system for a holiday company, you will hear the stakeholders talk about reservations, bookings, and so on, but the single most important abstraction, `Order`, may never be mentioned explicitly if it does not exist in current business systems. You generally know when you have found a hidden class because the whole model seems to gel suddenly with the introduction of this single, new abstraction. This happens surprisingly often—in fact, if we're ever having trouble with an analysis model and it just doesn't seem to be making sense, we go on a search for hidden classes. If nothing else, this makes us ask some penetrating questions and improves our understanding of the problem domain.

8.4.1.1 Noun/verb analysis procedure

The first step in noun/verb analysis is to collect as much relevant information as possible. Suitable sources of information are

- the requirements model;
- the use case model;
- the project glossary;
- anything else (architecture, vision documents, etc.).

After collecting the documentation, analyze it in a very simple way by highlighting (or recording in some other way) the following:

- nouns – for example, flight;
- noun phrases – for example, flight number;
- verbs – for example, allocate;
- verb phrases – for example, verify credit card.

Nouns and noun phrases may indicate classes or class attributes. Verbs and verb phrases may indicate responsibilities of classes.

If you come across any terms that you don't understand during this process, seek immediate clarification from a domain expert and add the term to the project glossary. Take the list of nouns, noun phrases, verbs, and verb phrases and use the project glossary to resolve any synonyms and homonyms. This creates a list of candidate classes, attributes, and responsibilities.

Once you have this list of candidate classes, attributes, and responsibilities, you make a tentative allocation of the attributes and responsibilities to the classes. You can do this by entering the classes into a modeling tool and adding the responsibilities as operations to the classes. If you have found any candidate attributes, then you can tentatively assign these to classes as well. You might also have gained some idea of relationships between certain classes (the use cases are a good source of these), so you can add some candidate associations. This gives you a first-cut class model that you can refine by further analysis.

8.4.2 Finding classes by using CRC analysis

CRC is a brainstorming technique in which you capture on sticky notes the important things in the problem domain.

A very good (and fun) way to get user involvement in finding classes is to use CRC analysis—CRC stands for class, responsibilities, and collaborators. This technique uses the world's most powerful analysis tool, the sticky note! So popular is the CRC method that there is a (possibly apocryphal) story that at one point a company actually marketed sticky notes already marked out with class name, responsibilities, and collaborators.

You begin by marking up some sticky notes as shown in Figure 8.4. The note is divided into three compartments. In the top compartment you record the name of the candidate class; in the left compartment, the responsibilities; and in the right, the collaborators. Collaborators are other classes that may collaborate with this class to realize a piece of system functionality. The collaborators compartment provides a way of recording relationships between classes. Another way to capture relationships (which we prefer) is to stick the notes on a whiteboard and draw lines between the collaborating classes.

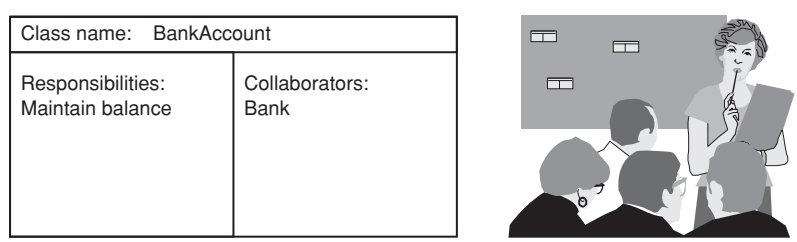


Figure 8.4

8.4.2.1 CRC analysis procedure

CRC analysis should always be used in conjunction with noun/verb analysis of use cases, requirements, glossary, and other relevant documentation, unless the system is very simple. The CRC analysis procedure is straightforward

and the key is to separate information *gathering* from information *analysis*. CRC is therefore best run as a two-phase activity.

8.4.2.2 Phase 1: Brainstorm – gather the information

Stakeholder involvement is essential for CRC success.

The participants are OO analysts, stakeholders, and domain experts. You also need a facilitator. The procedure is as follows.

1. Explain that this is a true brainstorm.
   - **1.1.** All ideas are accepted as good ideas.
   - **1.2.** Ideas are recorded but *not* debated – never argue about something, just write it down and then move on. Everything will be analyzed later.
2. Ask the team members to name the "things" that operate in their business domain – for example, customer, product.
   - **2.1.** Write each thing on a sticky note – it is a candidate class, or attribute of a class.
   - **2.2.** Stick the note on a wall or whiteboard.
3. Ask the team to state responsibilities that those things might have – record these in the responsibilities compartment of the note.
4. Working with the team, try to identify classes that might work together. Rearrange the notes on the whiteboard to reflect this organization and draw lines between them. Alternatively, record collaborators in the collaborators compartment of the note.

8.4.2.3 Phase 2: Analyze information

Important business concepts generally become classes.

The participants are OO analysts and domain experts. How do you decide which sticky notes should become classes and which should become attributes? Go back and look at Section 8.3.2—analysis classes *must* represent a crisp abstraction in the problem domain. Certain sticky notes will represent key business concepts and clearly need to become classes. Other notes may become classes or attributes. If a note logically seems to be a *part* of another note, this is a good indication that it represents an attribute. Also, if a note doesn't seem to be particularly important or has very little interesting behavior, see if it can be made an attribute of another class.

If in doubt about a note just make it a class. The important point is to make a best guess and then drive this process to closure; you can always refine the model later.

8.4.3 Finding classes by using the RUP stereotypes

According to RUP, it can be helpful to look for **«boundary»**, **«control»**, and **«entity»** classes.

A useful technique comes from RUP in the form of RUP stereotypes. The idea is that you consider three distinct types of analysis class during your analysis activity. This is a way of focusing your analysis on specific aspects of the system. We consider this an optional technique that you can use to complement the core noun/verb and CRC card analysis techniques presented earlier.

Three distinct types of analysis class can be distinguished by the stereotypes shown in Table 8.1.

We look at how to find each of these types of analysis class in the next three subsections.

Table 8.1

Stereotype	Icon	Semantics
«boundary»		a class that mediates interaction between the system and its environment
«control»		a class that encapsulates use-case-specific behavior
«entity»		a class that is used to model persistent information about something

8.4.3.1 Finding «boundary» *classes*

These classes exist on the boundary of your system and communicate with external actors.

You find these classes by considering the subject (system boundary) and discovering what classes mediate between the subject and its environment. According to RUP there are three types of «boundary» class:

1. user interface classes – classes that interface between the system and humans;
2. system interface classes – classes that interface with other systems;
3. device interface classes – classes that interface with external devices such as sensors.

Each communication between an actor and a use case in your model must be enabled by some object in your system. These objects are instances of boundary classes. You can work out what type of boundary class is indicated by considering what the actor represents (Table 8.2).