CSci 658: Software Language Engineering
Object Oriented Software Development

H. Conrad Cunningham

26 April 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of April 2018 is a recent version of Firefox from Mozilla.

Object-Oriented Software Development

Introduction

TODO: Add

Simplified Software Development Lifecycle

For the purposes of this discussion, assume that software development proceeds through the four lifecycle phases shown in the following diagram.

```
.__________________.
|                  |
|     Analysis     |<--:
|__________________|   |
         |             |
         |             ^
._________V_________.   |
|                  |-->:
|      Design      |   |
|__________________|<--.
         |             |
         |             ^
._________V_________.   |
|                  |-->:
|  Implementation  |   |
|__________________|<--:
         |             |
         |             |
._________V_________.   ^
|                  |   |
|   Maintenance    |-->:
|__________________|
```

Analysis

In the *analysis* phase we move from a vague description of the problem to be solved to a precise and unambiguous *requirements specification*.

The requirements specification might be a precise, but informal description written in careful natural language text and diagrams. Alternatively, the specification might be a formal description written in a mathematically precise language. Or the requirements specification might be something in between these.

The requirements specification should be:

- complete
- consistent
- readable by application domain experts and software developers
- testable
- independent of programming considerations

Design

In the *design* phase, we move from a requirements specification to a *design specification*. The design specification gives the system structure. The design tasks are to:

- Break the programming task into manageable parts.
- Define the relationships among the parts.
- Incorporate any required or appropriate pre-existing components.
- Keep the design independent of implementation language and hardware *details*. (We can, however, use knowledge of high-level architectures and language paradigms.)

Implementation

In the *implementation* phase, we move from a design specification to a *tested executable system*. The implementation tasks are to:

- Translate the design into working software on appropriate hardware
- Use the details of the programming language carefully

Maintenance

In the *maintenance* phase, we move from a complete "working" system to a *modified system*. The maintenance tasks are to:

- Repair any *errors* in analysis, design, or implementation that have been found
- Adapt to the *changes* in requirements that have occurred

We observe the following.

- In successful software systems, the maintenance phase is more important than implementation.
- Often more than half of the development effort will occur during maintenance.

We conclude that we should:

- Do initial analysis, design, and implementation *very carefully*
- *Design for change!*

"Programming in the Small" and "Programming in the Large"

The type of software development projects familiar to most students can be described as *programming in the small.* Such projects have the following attributes:

- Programs are developed by a single programmer or perhaps a small group of programmers.
- All aspects of the project can be understood by a single individual.
- The major problem is the development of the algorithms and data structures needed to solve the task at hand.

Programming in the large characterizes projects with the following attributes:

- The software system is developed by a large team of programmers, often with considerable specialization.
- No single individual can (likely) understand all aspects of the project.
- The major problem is the coordination of the diverse aspects of the project–people and software systems.

The techniques of object-oriented design and programming are useful in both programming in the small and programming in the large situations. However, some of the techniques are best appreciated when the difficulties of programming in the large are understood.

Object Orientation

In contemporary practice, most software engineers approach the design of programs from an object-oriented perspective.

Key idea (notion?) in object orientation: The real world can be accurately described as a collection of objects that interact.

This approach is based on the following *assumptions*:

1. Describing large, complex systems as interacting objects make them *easier to understand* than otherwise.
2. The *behaviors* of real world objects tend to be *stable* over time.
3. The different *kinds* of real world objects tend to be *stable.* (That is, new kinds appear slowly; old kinds disappear slowly.)
4. *Changes* tend to be *localized* to a few objects.

Assumption 1 simplifies analysis, design, and implementation–makes them more reliable.

Assumptions 2 and 3 support reuse of code, prototyping, and incremental development.

Assumption 4 supports design for change.

The object-oriented approach:

- Uses the same basic entities (i.e., objects) throughout the lifecycle
- Identifies the basic objects during analysis
- Identifies lower-level objects during design, reusing existing object descriptions where appropriate
- Implements the objects as software structures (e.g., Java classes).
- Maintain the object behaviors.

An Object Model

See the sections on the Object-Oriented and the related Prototype-Based paradigms in the Programming Paradigms notes.

Requirements Analysis

The task of the analysis phase is to define the problem and write a clear, consistent, complete, and precise description of the system to be constructed. The analysts must elicit the requirements for the system from the clients and communicate the requirements to the system designers, who have the task of designing the new system.

The analysts must determine the scope of the system: What must the system accomplish? And, perhaps just as importantly, what behaviors are clearly outside the system in its environment?

In approaching the requirements analysis for a new system, the analysts should do the following.

- Read *all* existing documents that potentially describe the requirements for the new system.

  This includes all documents that directly address the requirements as well as documents that may indirectly address them. The indirect sources include such items as memos, meeting minutes, problem reports from the (manual or automated) system being replaced, and employee, supplier, or customer complaints and suggestions. The indirect sources might also include information about how recent or expected future changes in the business, regulatory, social, or technological environment may impact the requirements for the system.

- Examine system outputs carefully.

  This includes the outputs (e.g., reports) generated in the current system and the descriptions of outputs desired from the new system.

- Interview users.

  The analysts should talk to a wide range of experienced users of the current system and its outputs and take careful notes. The experienced users are the ones who know how the system is "really" being used in practice – what its strengths and weaknesses are, what works well and what must be worked around, who uses what aspect of the system or its outputs, etc.

  The users of the system may have many different relationships to the system and, hence, will likely have different perspectives on how it works. The users might include a wide range of employees of the organization (clerical personnel, computer operators, managers, technical professionals, factory workers, salespersons, etc.) and might also include customers or suppliers to the client organization.

- Examine documentation of the current system.

  The analysts should review the official documentation on the current system (whether automated or manual) as well as any unofficial or personal notes users or maintainers of the system have.

Good analysts are good detectives! Among the mass of detail, the analysts must find the clues that allow them to solve the mystery of what system the client needs. It is important that analysts keep a complete record of the information they have gathered and their reasoning on any "conclusions" they reach about that information.

The result of the analysts' work is a document called the *requirements specification*. Typically this will be a natural language (e.g., English) document. The writers of this document should use great care, using the language in a clear, consistent, and precise manner. The writers are establishing the vocabulary for communication with the clients and among the designers, implementers, and testers of the system.

Note: The accompanying slide set Using CRC Cards discusses the processes of object-oriented requirements (this subsection) and design (next subsection) using methods built around Class-Responsibility-Collaboration (CRC) cards.

Object-Oriented Design

The goals of the design phase are to:

- Identify the *classes*.

  What kinds of objects do we have?

- Identify the *responsibilities* (i.e., functionality) of each class.

  What does each kind of object do? What information does it hold?

- Identify the *collaborations* of each class (i.e., the relationships among the classes).

  Does one kind of object use another in some way? Is it a special case of another kind?

The actual design process is iterative. Elaboration of a class may lead to identification of additional classes or changes to those already identified.

Classes should be *crisply* defined. It should be as easy as possible to determine what class an object belongs in. Coming up with a good categorization is often difficult in complex situations.

The responsibilities of a class are of two types:

1. to carry out some action – that is, an *operation*.
2. to hold some key information – that is, an *attribut*e (part of the state).

Responsibilities should be defined precisely. Ambiguity and imprecision will likely cause problems later in the design and implementation.

The collaboration relationships among classes should not be excessively complex. A collaboration between two classes means that one class depends upon the other in some way. A change in one class may necessitate a change in the other.

The information gathered in the design phase is the basis for the implementation. A good design makes the implementation easier and faster and the resulting product more reliable.

The basic methods for identifying classes and responsibilities are as follows.

- ***To find objects and their classes*:** Begin with the *nouns* in the requirements specification.
- ***To find responsibilities*:** Begin with the *verbs* in the requirements specification.

Finding Classes and Responsibilities

As an example, consider the following telephone book example: *You are to build an automated telephone book system for a university. The telephone book should contain entries for each person in the university community–student, professor, and staff member. Users of the directory can look up entries. In addition, the administrator of the telephone book can, after supplying a password, insert new entries, delete existing entries, modify existing entries, print the telephone book to a printer, and print a listing of all students or of all faculty. The entries in a listing are to be arranged in alphabetical order by family name.*

To develop an object-oriented design model for this application, as designers we can carry out the following steps:

1. **Identify the candidate *classes*.**

   Begin by listing the nouns, noun phrases, and pronoun antecedents from the requirements specification, changing all the plurals to singular.

   In the telephone book example, these include: *you (the designer), automated telephone book system, university, telephone book, entry, person, university community, student, professor, staff member, employee, user, directory, administrator, password, printer, listing, faculty, alphabetical order, family name*

   Other classes may be be implicit in the specification or may emerge as the design of individual classes proceed and the designer's knowledge of the application domain increases. For example, the design may need classes corresponding to:

   - the "subjects" (i.e., actors) of passive voice sentences. To find these, a designer may need to restate the passive sentence as an equivalent active sentence.
   - parts of items named. For example, the *name*, *address*, and *telephone number* fields of the *entries* are not explicitly mentioned, but they are implicit in what is commonly meant by telephone book entries.
   - known interfaces to the environment of the system. For example, user interface entities such as a *menu* are implied by the need for some way for a user to interact with the application.
   - data structures for storing the *collection* of personal entries that make up the telephone book.

   Several, but not necessarily all, of these will become classes in our design.

2. **Identify the candidate *responsibilities*.**

   Begin by listing the verbs from the specification. Listing the objects of transitive verbs may also be helpful.

In the telephone book example, these include: *build, contain (entry), look (up entry), supply (password), insert (new entry), delete (existing entry), modify (existing entry, print (telephone book), print (all students), print (all employees), set (telephone number field), get (telephone number field), compare (entries), to be arranged*

Transitive verbs become operations that respond to inputs to the object. All of the above verbs except "to be arranged" are transitive and, hence, will likely give rise to operations.

Intransitive verbs typically specify attributes of classes. For example, "to be arranged in alphabetical order" in the above example denotes an ordering property of the entries in a listing. It is not an operation. Of course, in some cases, this kind of attribute might require a "sort" operation.

3. **Eliminate classes and responsibilities that are outside of the system scope.**

   At this point, we may want to narrow the list of candidate classes by quickly dividing them into three categories based on their relevance to the system scope:

   a. **critical classes** (i.e., the "winners"), which we will definitely continue to consider. These are items that directly relate to the main entities of the application.

      In the telephone book example, these might include *telephone book, entry, student, faculty, staff member*, and so forth.

   b. **irrelevant candidates** (i.e., the "losers"), which we will definitely eliminate at this point. These are items that are clearly outside the system scope.

      In the telephone book example, these might include *university* which we do not need to explicitly model in this application) and *printer* (which is handled by other software/hardware outside of this application).

   c. **undecided candidates** (i.e., the "maybes"), which we will review further for categorization. These are items that we may not be able to categorize without first clarifying the system boundaries and definition

      In the telephone book example, these might include the *automated telephone book system*, an item that definitely will be built (it is the whole system) but for which we may not need an explicit class.

4. **Combine synonym classes and synonym operations into single abstractions.**

   For example, *automated telephone book*, *telephone book*, and *directory* can be combined into a single `PhoneBook` class.

Similarly, *entry* and *person* can be combined into a single `Person` class.

In addition, all the `print` operations probably could be combined into a single `print` operation with the differing functionalities specified by the parameter.

5. **Distinguish attributes from classes.**

   Some candidate classes may turn out to represent information held by other classes instead of being classes themselves.

   A candidate class may be an attribute (i.e., a responsibility) of another class rather than itself a class if:

   - it does not do anything – i.e., it has no operations.
   - it cannot change state.

   For example, we might choose to make the entity *name* an immutable string and make it an attribute of a class `Person` rather than have it a separate class.

6. **Be wary of adjectives.**

   Adjectives modify nouns. In our technique, nouns give rise to classes or objects.

   - An adjective might be *superfluous* as far as our object analysis is concerned. For example, the adjective "automated" in "automated telephone book" adds no extra information to "telephone book" because the adjective was implicit in the context of the specification anyway.
   - An adjective may signal the need for an *attribute* of the associated class. For example, the phrase "red truck" might imply the need for a color attribute of a `Truck` class–unless, of course, only one color of truck is needed in the application.
   - An adjective may signal the need for a *subclass*. For example, the phrase "fire truck" would likely call for a subclass `FireTruck` of class `Truck` if other kinds of trucks are needed in the application. A "fire truck" has several behaviors and attributes that differ from the generic concept of a "truck".
   - An adjective might just indicate the need for an *instance* of a class. For example, in a context where there is exactly one "red truck" and where "red" trucks need no different behaviors than any other trucks, it is sufficient for the "red truck" to simply be an instance of class `Truck`.

   Prepositional phrases may modify nouns in the requirements specification. Such phrases may lead to subclasses, objects, or instances as described above for adjectives.

Adverbs may modify adjectives in the requirements specification. They provide other information about the adjective that should be considered in the analysis. For example, a reference to a "brilliantly red truck" might signal the need for another attribute, subclass, or instance that is different from a plain "red truck" or from a "dull red truck".

7. **Consider architectural design issues.**

   - **Identify *hot spots.*** Structure classes and collaborations accordingly.

     A hot spot is a portion of the system that is likely to change from one system variant to another.

     To readily support change, encapsulate the variable aspects within components and design the interfaces of and the relationships among system components so that change to the architecture is seldom necessary.

     This technique enables convenient reuse of the relatively static, overall system architecture and common code. It makes change easier to implement at hot spots.

     In the telephone book example, we might consider aspects of the application that likely will need to change over time to meet the needs of the identified client. The hot spots identified might be how the telephone book information is stored, how the printed listings are formatted, what the user interface looks like, what exact information is stored in entries, what kinds of computing platforms the application executes on, what types of printers are used, etc. The detailed design decisions relative to these issues should be encapsulated within single components and documented well.

     Sometimes we need to approach the design of an application as the design of a whole *product line* rather than the design of a single product.

     For example, we might consider what might change if we needed to modify the application to handle the needs of other organizations. In the telephone book example, what would need to change if we wanted to make it useful to other universities? to government agencies? to private nonprofit service agencies? to businesses? to institutions in other countries (i.e., internationalization)? Clearly, the categorization of student, staff, and professor would need to be flexible. The handling of the name and address data and of other aspects of the `Person` entries would need to be flexible.

     Similarly, we can consider what might need to change if we wanted to expand the application from just maintaining telephone book information to other types of membership applications. Can we separate the application-specific behaviors from more general behaviors? Can we make it so that it is easy to design new specific behaviors and plug them in to the overall application structure?

- **Use appropriate *design patterns* to guide structuring of the system.**

  A design pattern is a design structure that has been successfully used in a similar context–i.e., a reusable design.

  Design patterns may be distillations of the development organization's experience – or may be well-known general patterns selected from a catalog such as *Design Patterns: Elements of Reusable Object-Oriented Software* by the "Gang of Four" [Gamma 1995].

  The use of the design pattern may require the addition of new classes to the design or the modification of core classes.

  An example of a high-level pattern is the Pipes and Filters pattern. In Unix for instance, a filter is a program that reads a stream of bytes from its standard input and writes a transformed stream to its standard output. These programs can be chained together with the output of one filter becoming the input of the next filter in the sequence via the pipe mechanism. Larger systems can thus be constructed from simple components that otherwise operate independently of one another.

  An example of a lower level pattern is the Iterator. This pattern defines general mechanisms for stepping through container data structures element by element. For instance, a Java object that implements the `Enumeration` interface is returned by the `elements()` of a `Vector` object; successive calls of the `nextElement()` method of the `Enumeration` object returns successive elements of the `Vector` object.

- **Take advantage of existing software *frameworks.***

  A framework is a collection of classes–some abstract, some concrete–that captures the architecture and basic operation of an application system. Systems are created by extending the given classes to add the specialized behaviors.

  Frameworks are "upside down libraries" – system control resides in framework code that calls "down" to user-supplied code.

  Examples of software frameworks include graphical user interface toolkits like the Java AWT and some discrete event simulation packages.

  To fit an application into a framework may require addition or modification of core classes.

8. **Associate the operations with the appropriate classes.**

For example, in the telephone book example associate `lookup`, `insert`, `delete`, and `modify` entry operations with the `PhoneBook` class, associate `compare`, `setPhoneNumber` and `getPhoneNumber` with the `Person` class, etc.

Sometimes there might be a choice on where to associate an operation. For example, the "insert person into telephone book" operation could be a operation of `Person` (that is, insert *this* person into the argument telephone book) or a operation of `PhoneBook` (that is, insert the argument person in *this* telephone book).

Which is better? Associating the operation with `Person` would require the `Person` class to have access to the internal representation details of the `PhoneBook`. However, associating the operation with `PhoneBook` would not require the `PhoneBook` to know about the internal details of the `Person` class–except to be able to `compare` two entries. Thus making `insert` an operation of `PhoneBook` and `compare` an operation of `Person` would best maintain the encapsulation of data.

Principle: *An object cannot manipulate the internal data of another object directly; it must use an operation of that object.*

During design we should not be concerned with the minute details of the implementation. However, it is appropriate to consider whether there is a "reasonable" implementation. In fact, it is better to make sure there are two different possible implementations to ensure flexibility and adaptability.

It is good to have more than one person involved in a design. A second designer can review the first's work more objectively and ask difficult questions – and vice versa.

Coming Up with Names

The selection of names for classes and operations is an important task. Give it sufficient time and thought.

- Names of classes should be singular nouns. In Java and Scala, these names should normally begin with an uppercase letter.
- Names of responsibilities (operations) should be verbs or short sequences of words containing one verb. In Java and Scala, these names should normally begin with a lowercase letter.
- Names of booleans should indicate meaning of the *true* value.
- Names should be easily recognized and understood by domain experts.
- Names should be short.
- Names should be pronounceable (read them out loud).

- Names should be consistent within the project (perhaps the entire development organization).
- Names should be unambiguous, not having multiple interpretations. Use abbreviations with care.
- Names should use capitalization and underscores, but avoid digits.

Finding Relationships Among Classes

Two classes in a design may be related in one of several ways. The three relationships that are common are:

- *use* or awareness (*uses*)
- *aggregation* (*has-a*) – sometimes called containment or composition
- *inheritance* (*is-a*) – sometimes called generalization, extension, or specialization

Use Relationship

Class A *uses* B when:

- an operation of A receives/returns an object of B
- an operation of A must examine or create an object of B
- an object of A "contains" objects of B (as instance variables)

If objects of A can carry out all operations without any awareness of B, then A *does not use* B.

From the telephone book example, `PhoneBook` uses `Person` objects (that is, it inserts, deletes, modifies, etc., the entries). `Person` objects do not use `PhoneBook` objects.

If class A uses class B, then a change to class B (particularly to its public interface) may necessitate changes to class A. The following principle will make modification of a design and implementation easier.

Principle: *Minimize the coupling between classes* (that is, the number of classes used by a class.)

Aggregation Relationship

Class A uses class B for *aggregation* if objects of A contain objects of B.

Note: Object A *contains* (*has an*) object B if A has an *instance variable* that somehow designates object B–a Java reference to B, the index of B, the key of B, etc.

Aggregation is a one-to-N relationship; one object might contain several others.

Note: Aggregation is a special case of the use relationship. If asked to identify the aggregation and use relationships, identify an aggregation relationship as such rather than as a use relationship.

For example, objects of the `Person` class in the telephone book example contain (*has*) `name`, `address`, and `phoneNumber` objects.

The Pascal `record` and C `structure` are aggregations; they contain other data fields. They are not, however, objects.

Inheritance Relationship

Class D *inherits from* class B if all objects of D are also objects of B.

As noted previously,

- all B operations must be valid for D objects (perhaps implemented differently),
- D objects can have additional operations and data fields.

In the telephone book example, we design `Professor` to inherit from `Employee`; similarly, we design `Staff` to inherit from `Employee`. We make `Employee` itself inherit from `Person`; similarly, we design `Student` to inherit from `Person`.

Inheritance can lead to powerful and extensible designs. However, use and aggregation are more common than inheritance.

Object-Oriented Implementation

The outputs of the object-oriented design phase include:

- descriptions of each class
- descriptions of each operation (i.e., method)
- diagrams giving relationships among the classes

The purpose of the implementation phase is to code, test, and integrate the classes into the desired application system.

Object-oriented development involves an integration of testing with implementation:

- Implement and test each class or cluster of closely related classes separately.
- Once each cluster is working correctly, integrate it into the program.
- Perhaps defer some operations until later–build a prototype rapidly to explore requirements.

- If done carefully, incrementally evolve the prototype into a fully functional system.

Note: If we seek to evolve a prototype into a full program, we should never hesitate to reopen the analysis or design if new insight is discovered.

Conclusions

TODO: Add

Exercises

TODO: Add

Acknowledgements

In Fall 2016 and Spring 2017, I adapted these lecture notes from my previous notes on this topic. The material from my previous notes is based, in part, on the presentations in the following books:

- David Bellin and Susan Suchman Simone. *The CRC Card Book*, Addison Wesley, 1997.
- Timothy Budd. *Understanding Object-Oriented Programming with Java*, Updated Edition, Addison Wesley, 2000.
- Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*, Addison Wesley, 1995.
- Cay S. Horstmann. *Mastering Object-Oriented Design in C++*, Wiley, New York, 1995.
- Pete Thomas and Ray Weedom. *Object-Oriented Programming in Eiffel*, Addison-Wesley, Workingham, UK, 1995.
- Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*, Prentice-Hall, 1990.

I wrote the first version of these notes in Spring 1996 for my CSci 490 Special Topics course on object-oriented design and programming using C++. I expanded the notes for the first Java-based version of CSci 211 (then titled File Systems) during Fall 1996.

I revised the notes incrementally over the next decade for use in my Java-based courses on object-orientation, software architecture, and computer simulation.

I partially revised the notes for use in my Scala-based classes beginning in Fall 2008 and Lua-based classes beginning in Fall 2013.

For the CSci 450 Programming Languages class in Fall 2016, I moved the discussion of the Object Model from the OO notes and combined it with the discussion of programming paradigms from my Functional Programming notes as part of a new introductory document for that course. (See the notes on the Object-Oriented programming paradigm in Chapter 1 of that document.) However, for Spring 2018, I separated the discussion of programming paradigms from the larger chapter into a separate document Programming Paradigms.

In Summer 2017 and Spring 2018 I reformatted these notes to use Pandoc Markdown, improved the presentation in a few places, and linked it into the other documents.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

References

[Bellin 1997] : David Bellin and Susan Suchman Simone. *The CRC Card BookUs*Us, Addison Wesley, 1997.

[**Budd 2000**] Timothy Budd. *Understanding Object-Oriented Programming with Java*, Updated Edition, Addison Wesley, 2000.

[**Fishwick 1995**] Paul A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*, Addison Wesley, 1995.

[**Gamma 1995**] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[**Horstmann 1995**] Cay S. Horstmann. *Mastering Object-Oriented Design in C++*, Wiley, 1995. (Chapters 3-6 on "Implementing Classes", "Interfaces", "Object-Oriented Design", and "Invariants" especially influenced my views on object-oriented design and programming.)

[**Thomas 1995**] Pete Thomas and Ray Weedom. *Object-Oriented Programming in Eiffel*, Addison-Wesley, Workingham, UK, 1995.

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

[**Wirfs-Brock 2003**] Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, 2003.

Concepts

TODO: Add