A Road Map For OOA and OOD

1. Why are we here?

In this class, you have been hit with a bewildering array of new terminology. Words like "class", "instance", "method", "superclass", "design", are all being thrown around with reckless abandon. It's easy to lose your place. The purpose of this road map is to help keep you grounded during the process of designing your object-oriented program. It will explain the purpose of each step in the system, and give some brief examples of good practice. The goal is for you to have some guidance as you go about designing and building your first object-oriented programs.

1.1 Keeping the goal in view -- Why OOP?

Object-oriented program has become the dominant programming style in the software industry over the last 10 years or so. The reason for this has to do with the growing size and scale of software projects. It becomes extremely difficult to understand a procedural program once it gets above a certain size. Object-oriented programs scale up better, meaning that they are easier to write, understand and maintain than procedural programs of the same size. There are basically three reasons for this:

1. Object-oriented programs tend to be written in terms of real-world objects, not internal data structures. This makes them somewhat easier to understand by maintainers and the people who have to read your code -- but it may make it harder for you as the initial designer. Identifying objects in a problem is a challenge

2. Object-oriented programs encourage *encapsulation* -- details of an objects implementation are hidden from other objects. This keeps a change in one part of the program from affecting other parts, making the program easier to debug and maintain. This does take some getting used to -- past students have called it "hypertext programming" because your program is spread out among a variety of objects.

3. Object-oriented programs encourage modularity. This means that pieces of the program do not depend on other pieces of the program. Those pieces can be reused in future projects, making the new projects easier to build.

Because of the overhead costs referred to above, OOP is probably not the best choice for very small programs. If all you want to do is sum up file sizes in a directory, you probably don't want to have to identify objects, create reusable structures and all that. Since, the programs that you write for this class are necessarily small (so you can do them in the quarter), so it may not always be obvious where the big benefit from using objects is.

A good object-oriented program, then, is one that takes advantage of the strengths of object-oriented programs as listed above. Some top-level guidelines for doing this include:

- Objects in your system should usually represent real-world things. Talk about cash registers and vending machines and stop lights, not linked-lists, hashtables or binary trees.
- Objects in your system should have limited knowledge. They should only know about the other parts of the system that they absolutely have to in order to get their work done. This implies that you should *always* have more than one object in the system.

It's not always clear, however, how to implement goals like this in practice. That's where the design comes into play.

1.2 So it won't crash on your head: Why Design?

In order to get the most out of using objects, OO programs require some kind of design work before programming starts. You, as the programmer, are much more in the position of an architect. Why do architects spend so much time on blueprints and models? One reason is that the blueprint allows the architect to solve the problem in a relatively low-cost environment before actually using brick and concrete. The final building is better: safer, more energy-efficient, and so on, because the worst mistakes are corrected on paper, cheaply and quickly, before a building falls on somebody's head. As a nice side effect, the precision of the blueprint makes it a useful tool for communicating the design to the builders, saving time in the building process.

OO probably requires more up-front design work than procedural programs because the job of identifying the objects and figuring out how to divide the tasks among them is not really a programming job. You really do need to have that started before you sit down to hack -- otherwise, how will you know where to start?

Similarly, time spent getting the design right before you start programming will almost always save you time in the end. It's much, much easier to make major changes on a design, which is after all just squiggly lines on paper, then it is to make changes in hundreds or thousands of lines of code.

1.3 Several Activities to a Brilliant Design

The design process is typically split into two distinct phases: Object-Oriented Analysis (OOA) and Object Oriented Design (OOD). Yes, I know that it is confusing to have one of the sub-steps of the design process also called "design". However, the term is so entrenched that it's hopeless to start creating new jargon.

We'll call these things "activities" rather than "steps" to emphasize that they don't have to be done in any particular order -- you can switch to another activity whenever it makes sense to do so. When you are actually doing this, you'll find that you want to go back and forth between OOA and OOD repeatedly. Maybe you'll start at one part of your program, and do OOA and OOD on that part before doing OOA and OOD on another part. Or maybe you'll do a preliminary OOA, and then decide while working on the OOD that you need more classes, so you jump back to OOA. That's great. Moving back and forth between OOA and OOD is the best way to create a good design -- if you only go through each step once, you'll be stuck with your first mistakes all the way through. I do think that it's important, though, to always be clear what activity you are currently doing -- keeping a sharp distinction between activities will make it easier for you to make design decisions without getting tied up in knots.

In the OOA phase, the overall questions is "What?". As in, "What will my program need to do?", "What will the classes in my program be?", and "What will each class be responsible for?". You do not worry about implementation details in the OOA phase -- there will be plenty of time to worry about them later, and at this point they only get in the way. The focus here is on the real world -- what are the objects, tasks and responsibilities of the real system?

In the OOD phase, the overall question is "How?". As in, "How will this class handle it's responsibilities?", "How can I ensure that this class knows all the information it needs?", "How will classes in my design communicate?". At this point, you are worried about some implementation details,

but not all -- what the attributes and methods of a class will be, but not down to the level of whether things are integers or reals or ordered collections or dictionaries or whatnot -- those are programming decisions.

One other thing about OOA and OOD. If you do any reading at all on the subject, you'll find that everybody that's ever written a book on object-oriented design has a slightly different idea about what goes in each step. The following is sort of a consensus view, based on student experiences, as well as the sources listed at the end.

2. What to do during OOA

There are three activities that you need to do as part of your object-oriented analysis.

1. Determine the functionality of the system, usually called requirements analysis.

2. Create a list of what classes are part of the system, and (just as important) what classes are not part of the system.

3. Distribute the functionality of the system among the list of classes. A piece of functionality assigned to a specific class is called a *responsibility*.

I'm not going to talk much about requirements analysis here. For the purposes of this class, it will mostly be done for you. The assignments you receive will lay out what the final program is required to do. In the real world, however, this is often the trickiest part of the software engineering process, as it is not always clear what the program is going to need to do. Typically what is done in an object-oriented requirements analysis is to create a list of *use cases*, small interactions that the system will have to support. That's outside our scope for now, so I'll focus on class creation and responsibilities.

2.1 Creating Classes

The goal of the class creation activity is to come up with a list of all classes that might possibly be part of your system, called *candidate classes*, and then determine which classes that you really want in the system, and which classes are outside it.

2.1.1 Where do Candidate Classes come from?

First and foremost, candidate classes come from brainstorming about the problem domain. Write down as many possible classes as you can think of, as quickly as you can.

Two things to keep in mind while brainstorming are:

- This is the beginning of your process, not the end -- don't worry about how any of this is going to be implemented yet. If it helps, pretend at this point that implementation is somebody else's problem

- Don't throw anything out. Two reasons. First, you never know when some crazy idea is going to wind up saving your project. Second, the classes that you don't include in the system are just as important for defining the bounds of your system as the ones that are inside.

Here are some good places to look for candidate classes:

- Go through your requirements document, and circle all the nouns.
- Think about the system. An object is a person, place or thing (usually).
    - Who are the people in the system? What roles do they play?
    - In what places does the system occur?
    - What tangible things are used in the system?
    - What transactions or sessions might need to be remembered?
- Are some of the things you've listed specific instances of a more general concept, or vice versa?
- Will some objects contain or be contained by other objects?

And here are some things to avoid

- Avoid computer-science terms like linked-list. You're trying to describe the world, not the program.
- Be suspicious of any object whose name ends in "er". Be very suspicious of any object that contains "manager". Frequently, these objects just mess with other objects data -- it's better for the other objects to mess with their own data.
- Don't include things you don't need even if they are in the real world. Everything is made of molecules, but you probably don't need a molecule class.
- Be suspicious of a class that represents the "whole thing." If your system is an elevator, and you have a box, doors, cables, buttons and controllers, you probably don't also need an object called "elevator". It's the sum of those things, not something separate. Only include a "whole thing" if the whole thing has data that is not in any of the component parts.

2.1.2 Choosing from Candidate Classes

Each time you are done brainstorming (you'll probably brainstorm more than once, as you continually refine your understanding of the system) go through the list of classes. You should be able to immediately split your list into three groups:

- Classes critical to the system that must be included
- Classes totally irrelevant to the system that must be rejected
- Classes that you are still undecided about

When in doubt, err towards being undecided.

Here are some questions to ask about candidate classes that you are undecided about:

- Does this class encapsulate something that would normally be done by the humans outside the system -- in that case, it's probably not part of the system (unless you're building a simulation).
- Is this class really identical to another class on the list?
- Does the class really have more than one possible state? Will it ever change state over the life of the program? If it only has one possible state, or if it doesn't change state then it's best included as an attribute of another class.
- Does the class have a singular name? A plural name might indicate that what you really want is the singular name, and some container class.
- Will the class have distinct responsibilities in the system?
- Will the class have unique knowledge in the system?
- Is this class needed by other classes in the system?
- Is this class really just an attribute of some other class?

2.2 Assigning Responsibilities

The other activity of OOA is assigning responsibilities to the classes that will carry them out. The tool for managing this activity is called a CRC card. CRC stands for Class, Responsibility, Collaborator, and describes the information on the card. Each class in the system has its own CRC card, which contains a list of all the responsibilities of that class, along with the other classes which collaborate in carrying out those responsibilities.

2.2.1 Where do responsibilities come from?

Most of the responsibilities in your system will initially come from you requirements document -- which is, after all, a list of all of the requirements in the system. Frequently, the combination of system requirements along with the classes in your system will imply further responsibilities. For example, a system responsibility to "sum the prices of items" might imply that an item class has the responsibility to "know my price".

Sometimes it is helpful to "role-play" -- pretending that you are each object in the system in turn, try to trace out interaction patterns to determine what each class will need to know and do.

Some other tips on creating responsibilities:

- Keep it simple. Each responsibility should be a short phrase -- not a sentence, and for sure not a paragraph.
- This is still a brainstorm activity, so the main goal is to just get things down on paper.
- Focus on what, not how -- implementation is still not your problem yet.

- Look for commonalities among classes -- you may be able to factor them out using inheritance.

2.2.2 How are responsibilities assigned?

Either you have a list of responsibilities and you are trying to assign them to classes, or you have a class and you are trying to define its responsibilities. Keep the following principles in mind:

- The anthropomorphic principle: an object is responsible for all the things that would be done to it in the real world.
- The expert principle: assign a responsibility to the class or classes that have the knowledge to perform it.
- If more than one object has the knowledge, then the two classes should collaborate on the responsibility.
- Distribute responsibility -- no one class should have most of the responsibilities. You shouldn't be able to point to one class as the "center" of your design.
- Each class should have some responsibility, if not, question whether it belongs there at all.
- If a class seems to have too much responsibility, or if its responsibilities seem unrelated, look for ways to split it into smaller classes.
- Rarely, a responsibility may become a class all by itself. Look for this if the responsibility is quite large, and potentially applies to a number of classes in your system that are not otherwise related.

2.3 OOA Checkpoint

Whenever you come out of an OOA phase, you should have a list of classes in your design, a list of rejected classes, and possibly a list of classes still under consideration. Each class in your design should have a list of responsibilities and collaborators that describe what the class does within the system.

Here are some characteristics of a good OOA:

- The classes are relatively small. At least some of them are general enough that you could imagine them being used again in a future project.
- Responsibility and control are distributed. The design has no explicit "center".
- Information with different rates of change are separated. It's almost always a bad idea for a class to combine relatively permanent information with relatively transient information. For example, in a sales database, you'd want to separate the customer's name and address (relatively permenanent) from the details of the last sale (transient).
- There are few assumptions about the language or system it will be implemented in.
- The OOA describes the world, not computer-science jargon.
- The objects in the system all have some responsibility.

- No object is merely a "manager" of another object's data.
- If the requirements were extended to include more things, the existing parts of the design would only change minimally.
- If the input or output methods of the program were to change radically, the existing design would only change minimally.
- There should be no redundancy.

3. What to do during OOD

The primary goal of the OOD phase is to convert your OOA into something that you could actually implement. In particular, the OOD is concerned with how information flows through your system. In this phase, you decide what information each class knows, and what other classes it needs to know about.

There are two activities in the OOD phase: the first is assigning attributes and services, the second is assigning links.

3.1 Assigning attributes and services

When you enter an OOD phase, you should have at least some of the responsibilities and collaborators for each class already assigned. At this point, you can start thinking about converting them into attributes and services.

Attributes represent the state of the object. The attributes are what the object "knows". Some hints on assigning attributes:

- An attribute should not contain the name of another object in the system. You do not need to create attributes to represent other objects that will also be denoted by links -- the link implies the attribute.
- Consider whether the attribute is best represented as its own class -- this is especially true if future changes may make the attribute more complex.
- Values that the object calculates from the other attributes are typically represented as services -- for example: calculateTotalPrice.
- Refer to a set of closely related values as one -- for example, **name** not **firstName, middleName** and **lastName** -- even if that may be how you implement it.
- Ask the following questions about an attribute:
    - Does it describe me? If so, add it to me.
    - Does it describe an object I know? If so, add it to the other object.
    - Does it describe something we share? If so, you might need an object to encapsulate the

interaction.

- Will every instance of this class need this attribute?
- Be suspicious of any attribute named "type" -- often, these can be implemented as a set of subclasses.
- It's often a good idea to make a distinction between attributes that will never (or rarely) change and attributes that will change frequently -- you might even want to give the transient attributes their own class (connected with a has-a). For example, if you were designing a physicians record system, *name* and *social security number* probably won't change, but *height* and *weight* probably will, and might deserve their own class.
- Here are things to think about attributes as you move toward implementation. Don't get too caught up in them at this point, but if there is other information about the attribute that you know, that's worth noting.
    - data type
    - unit of measure
    - legal range
    - is it required
    - default value

3.2 Assigning Services

A **service** represents something that the object knows how to do, and will typically be implemented as a method. Here are some things to think about when creating services:

- "Do it myself" -- an object typically does those things that are done to the real-world object it represents.
- Services should be performed by the object that has the information to do them -- that information should not be passed to other objects to make the calculation.
- Basic services, including get, set, add, remove, initialize and delete do not need to be shown in your model. You can consider them implied.
- Most services will come from the responsibilities already established for the class.
- Will every object of this class need this service?
- If a service contains the name of another object in the system, consider moving that service to that object -- or at least letting it help.
- Be suspicious of services that take as input another object in the system -- consider moving the service to that object.

3.3 Assigning Links

A link represents a connection between two classes, and means that those two classes need to share

information in some way. There are three kinds of links in an OO system:

An ***inheritance*** link (also called generalization/specialization, gen/spec, or subclass/superclass): This means that one of the classes does everything the other class does, plus a little more. Inheritance is represented by a semi-circle on the link, with the curved end towards the parent class. The sentence "<<subclass>> is a kind of <<superclass>> should be meaningful.

A ***composition*** link (also called part/whole, whole/part, or aggregation): This means that one of the classes is logically part of the other class, and that the two objects will always be connected. Composition links are represented by a triangle on the link, pointing toward the whole end. The sentence "<<whole>> has-a <<part>>" should be meaningful.

A ***broadcast*** link: This means that the two classes need to know about the other, but there is no further logical connection between them. The sentence "<<sender>> sends a message to <<receiver>>" should be meaningful.

Notice that a composition link and a broadcast link usually imply that the two classes actually know about each other (usually implemented as a data member), but an inheritance link does not imply a data connection.

Most of the connections in your system should be composition ones. They are easier to maintain and understand than inheritance connections. However, a good inheritance hierarchy can make the system implementation much simpler. Here are some guidelines on creating the various types of links.

3.3.1 When to create a composition link

A part/whole link should be created when it is logical to think of one class as being part of another class. Things to look for include:

- Instances of the part class will need to be created when the whole class is created, and will not need to be kept around after the whole is destroyed.
- Be sure that both sides of the part/whole do work and have data -- try to avoid the pattern "workaholic object -> data holder object."

3.3.2 When to create an inheritance link

Under some circumstances, it's helpful to create a general (abstract) superclass, and have the specific classes in your system be subclasses of it. Things to look for include:

- A number of classes have the same attribute set, and there is a sensible generalization
- A number of classes have the same services, and there is a reasonable generalization.

Other times, you will create a specialization of an existing class in your system. Signs that specializing might be helpful include:

- The class has an attribute that is only applicable to some of its instances. Consider making the instances that use that attribute a subclass.

- The class has a service that is only applicable to some of its instances. Consider making the instances that use the service into a subclass.

There are some questions you should ask yourself about any subclass/superclass link.

- The is-a rule: The sentence "<<subclass>> is a special kind of <<superclass>>" makes sense, and the sentence "<<subclass>> is a role played by <<superclass>>" does not. If the second sentence makes more sense, consider using a part/whole connection.

- The 100% rule: The subclass uses all of the functionality of the superclass. It's poor design to "block out" existing superclass functionality (even if the subclass might implement that functionality differently.

- The forever rule: An instance of a subclass or the parent class should always be classified as a member of that class. If you think an object might change types at some point in its life, you should try to separate the parts that will change from the parts that won't, and use composition to connect the pieces. For example, in an employee health benefits system, *RetiredWorker* should not be a sublclass of *Worker*, because instances of worker will need to become instances of *RetiredWorker*.

3.3.4 When to create a broadcast link

- One object needs to know information that another object has, but there is no other logical relation between them

- The existence of the objects in the system is independent of each other.

3.4 OOD Checkpoint

At the end of an OOD phase, you should have a class diagram that contains attributes, services and links between classes. Here are some questions to ask about your OOD at the end of an OOD phase:

- It's clear from the OOD how you would write your code.

- Every class should have at least one attribute or object connection. If not, consider moving that classes functionality to other classes.

- Every class should have at least one service. If not, reconsider whether the class is really needed.

- No object knows about every other object in the system, and no object is the clear "center" of the information flow.

- Objects connect to other objects **only** if they need information from them.

- There are no unnecessary middlemen -- if A needs to reach B, but can only do it through C, consider creating a direct connection between A and B.

- Attributes and services are as high in the inheritance hierarchy as possible.

Sources

In addition to student experience, there are a number of books on object-oriented design that went into the creation of this little tour. If you are interested in a more detailed look at the guidelines listed here, as well as some examples of how they are applied, you might want to check out these books.

David Bellin and Susan Suchman Simone, *The CRC Card BOOK.* Addison Wesley, 1997.

Peter Coad, *Object Models: Strategies, Patterns and Applications, Second Edition*. Yourdon Press, 1997.

Peter Coad, *Java Design: Building Better Apps and Applets*. Yourdon Press, 1997.

Martin Fowler, *UML Distilled: Applying the Standard Object Modeling Language*. Addison Wesley, 1997.

Craig Larman, *Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design.* Prentice Hall, 1998.