An Overview of the Main Course Themes

Steven J Zeil

August 20, 2013

Contents

This course is concerned with object-oriented programming (OOP), design (OOD), and analysis (OOA). Clearly, with so much of this OO stuff flying around, it would be nice if we had an idea of what an object is, and just what is meant by "object-oriented".

1 Leading Up to OO

1.1 Observations on Pre-OO Programming Languages

The Roots of OOP

- OO programming and design have their roots in simulation.
- Simula (1967) programming language introduced objects & classes to describe the physical world being simulated.

.....

OO programming and design have their roots in simulation. In the late 1960's, the population of programmers was dominated by people who had switched fields, typically from the mathematics or physical sciences, rather than people who had originally trained in computer science. Indeed, academic programs in computer science were far less common than they are today, and tended to treat CS as more of a tool to be applied to application areas rather than as a field of study in its own right. Commonly, a CS degree program would offer two tracks - business programming and scientific programming.

One of the more active and exciting application areas at the time was simulation. Programmers in this area began to recognize common patterns arising over and over in their code. A number of code libraries were designed and distributed that attempted to capture and simplify these common simulation tasks. Eventually, special-purpose simulation programming languages were developed.[1]

Simula (1967) is now the best known of those special purpose simulation languages. In what seemed "natural" to someone writign simulations, Simula allowed a programmer to organize a simulation program in terms of a world of objects that interacted with one another via programmed behaviors. What could seem more natural to someone writing a simulation of some part of the real world?

[1] There was ample precedence for this kind of specialization. The dominant languages of the time were generally perceived as tailored to specific applications: FORTRAN and APL for science applications, COBOL for business, LISP for A.I., SNOBOL for text/string processing, etc. Any of these languages *could* have been used for other kinds of applications, but it was certainly easier to work with a language that matched well with the application area.

1.1.1 Tension in Pre-OO Programming Languages

The history of programming languages and of design can be viewed as a continual contest against increasing complexity of software systems:

Problem: too many...	**Response**	
... statements	nesting ({...})	OK
	gather statements into functions	
... functions	nesting	(inadequate)
	gather functions into subsystems / "modules"	Too loosely defined
	gather functions & data into encapsulated ADTs	(Pre-dates OOPs by more than a decade)
... ADTs	Gather into namespaces/packages	Not much help
	Organize loosely into inheritance hierarchies	The OO approach

Meanwhile, back in the "main stream" of programming language developement, one of the main historic trends has always been trying to keep programs well organized even as programs get larger and larger. The earliest and simplest way to cope with size was to allow individual statements to be grouped together into functions/procedures/subroutines (all effectively meaning the same thing). In fact this idea pre-dates high-level programming languages, being implicit in the machine code instruction sets of almost all CPUs ever built. It may even pre-date any actual programming. Some attribute this idea to Ada Lovelace's programs for Babbage's never-constructed analytical engine.

Functions allow us to cope when the number of individual statements in a program grows so large as to be unwieldy. So what do we do when the number of functions grows too large to be manageable? Programmers and designers tried to cope by grouping functions into "modules" or "subsystems". These groupings were little more than a matter of documentation and physical arrangement of the functions within the source code files. On a team project, a single person who did not want to "play nice" could easily wreak havoc.

A more solid organizing principle arose in the notion of an Abstract Data Type (ADT). We'll review this concept in more detail in a later lesson, but an ADT groups together a data type name and a collection of functions for manipulating data of that type.

ADTs were something of a revolution in programming, as they often led to remarkably clean, reusable modules. Of course, like all modules, the ADT originally was implemented simply by documenting it as such: "the functions in this file provide the ADT Stack..." An uncooperative team member could often compromise this organization (usually giving plausible but short-sighted excuses such as "I could get the program done faster if I bypassed that" or "It runs a millisecond faster if I access that bit of data directly instead of using those functions").

The value of ADTs became widely accepted, and programming language designers responded by adding support for them into new programming languages. Languages such as Modula 2 (1970) and Ada (1983) added language mechanisms (called a “module” and a “package”, respectively) for grouping types and functions together. Both enforced “information hiding” by encapsulating portions of an ADT implementation that other programmers would then be unable to access without incurring compilation errors.

Many writers incorrectly regard ADTs, encapsulation, and information hiding as innovations of OOP, but in fact these ideas had been around for some time earlier, and were in widespread use before object-oriented programming languages (OOPL) made their entrance.

Indeed, we will eventually see that the vast majority of code written by even the very best programmers working in an OOPL is not particularly OO, but *will* exploit encapsulated ADTs very heavily.

1.2 Pre-OO Design Techniques

Observations on Pre-OO Design Techniques

- Google for ADTs.
  - Most of the examples you will get are general-purpose data structures (stacks, vectors, lists, etc)
  - It’s easy to get the impression that an ADT is something that someone else provides for you as a convenient little package of code.
  - Misses the big point
- ADTs are an *organizing principle* for your own programs.
- But how do you design a program in a way that winds up with good ADTs?

. . . . . .

OK, let’s agree that a good ADT is a wonderful thing. Just where do good ADTs come from? Or, to put it another way, when you are designing a program, how do you figure out which ADTs you should use?

If you search the web for the term "abstract data type", you will find lots of examples of things you might pull out of a library of data structures: stacks, queues, lists, etc. It’s easy to get the impression that an ADT is something that someone else provides for you as a convenient little package of code.

But that misses the point that "abstract data type" is an *organizing principle* for your own programs. ADTs for general purpose data structures are all very nice, but how do you come up with the special-purpose ADTs that organize your program in a way that is specific to the problem it solves?

Pre-OO design techniques were oriented towards functions performed by the program. For example, if you were working on automating the management of book handling in a typical library, you might start by dividing the system into major subsystems such as *RecordStorage*, *CheckInOut*, *AccountMgmt*, and *InventoryMgmt*. Under each subsystem, you would group functions to be performed, such as *CheckOutBook*, *CheckInBook*, and *RenewBook* under the *CheckInOut* subsystem, or *addBook*, *removeBook*, and *transferToBranch* under *InventoryMgmt*.

Stepwise Refinement & Top-Down Design

- The most fundamental design technique

. . . . . .

Stepwise Refinement Example

Automating book handling in a library:

- Might start by dividing the system into major subsystems such as `RecordStorage`, `CheckInOut`, `AccountMgmt`, and `InventoryM`
  Under each subsystem, you would group functions to be performed, such as `CheckOutBook`, `CheckInBook`, and `RenewBook` under the `CheckInOut` subsystem, or `addBook`, `removeBook`, and `transferToBranch` under `InventoryMgmt`.

. . . . . .

If pre-OO designers were asked to design one of these functions, say, *addBook*, they would probably do so by "stepwise refinement", a.k.a. "top-down design".

Stepwise Refinement Example II

They would reason that this function must both update the electronic card catalog so that the book can be found, and must also record that the book is present in the inventory of a particular branch:

```
function addBook (book, branchName)
{
```


```
  bookInfo = getBookInfo(book);
  record branchName as location in bookInfo;
  addToCardCatalog (bookInfo);
  addToInventory (bookInfo, branchName);
}
```

. . . . . .

Stepwise Refinement Example III

Next the designers would pick one of those rather vaguely understood lines of code in that function body and expand it, either in place or as a separate function. For example:

```
function addBook (book, branchName)
{
  bookInfo = getBookInfo(book);
  record branchName as location in bookInfo;
  // addToCardCatalog (bookInfo)
    add to AuthorIndex (bookInfo);
   add to SubjectIndex (bookInfo);
   add to TitleIndex (bookInfo);
  cardCatalog.addToAuthorIndex
  addToInventory (bookInfo, branchName);
}
```

Then, again, we would pick a vaguely understood line and expand it.

. . . . . .

Stepwise Refinement Example IV

```
function addBook (book, branchName)
{
  bookInfo = getBookInfo(book);
```

```
  record branchName as location in bookInfo;
  // addToCardCatalog (bookInfo)
   //   add to AuthorIndex (bookInfo);
    authorList = bookInfo.getAuthors();
   for each author au in authorList {
     catalog.authorIndex.addByAuthor (au, bookInfo);
   }
  add to SubjectIndex (bookInfo);
  add to TitleIndex (bookInfo);
  cardCatalog.addToAuthorIndex
  addToInventory (bookInfo, branchName);
}
```

• The process of " pick something vague and expand it" continues until the desired level of detail has been reached.

.....

Top-Down Example: Observations

- In this example, we have mentioned nearly a dozen functions. None of them have been associated with an ADT.
  - An experienced designer might spot some ADT candidates: Book, BranchLibrary, CardCatalog, etc.
  - But there's nothing in the *process* that encourages or supports this.

.....

Now, stepwise refinement is a valuable technique. It's still probably the technique of choice for low-level design (the design of an algorithm for a single function). But it's not nearly so useful as a technqiue for high-level design (the separation fo a large problem into separate modules). Consider that, at this point in our example, we have mentioned about a dozen different function names. *None* of them have been associated with an ADT. What are the odds that any of them would wind up inside an ADT? Pretty slim, usually.

An experienced designer might take note that the above process gives some hints as to some potential ADTs (*Book*, *branchLibrary*, *CardCatalog*, etc.). But this insight is pretty much separate from the design process itself. A designer has to be somewhat

schizoid, actively participating in the design while simultaneously standing aside as a detached observer, watching for hints of possible ADTs.

2 The Object-Oriented Philosophy

2.1 Program == Simulation

The Object-Oriented Philosophy

- Every program is really a simulation.
- The quality of a program's design is proportional to the faithfulness with which the structures and interactions in the program mirror those in the real world.

. . . . . .

If OO analysis and design is about any one thing, it's about how to find and recognize good ADTs. The first proponents of OO looked back to simulation, one of the early success stories in programming and design. Many early simulation programmers were able to construct programs that were organized in a way that made them easy to modify and maintain. This, it was thought, was because the focus on modeling real world, in a language that created a module for each kind of real-world object, led to programs that felt "natural" and easily understood by anyone who understood the part of the real world that was being simulated.

The *Object-Oriented philosophy* states:

- *Every program is really a simulation.*
- *The quality of a program's design is proportional to the faithfulness with which the structures and interactions in the program mirror those in the real world.*

According to this philosophy, a program to manage book handling in a library is really a simulation of what the librarians would have done prior to automation. A program to compute a company payroll is a simulation of what the accounting clerks would have done. A program to compute student grades at the end of a semester is a simulation of what a teacher would have done manually. Now, sometimes the worlds being simulated may be artificial or imaginary (e.g., chess and other games, fractal

mathematics) but, as long as they are well understood, that really doesn't matter. You may be able tome with exceptions to the "every program is a simulation" rule, but the very fact that you have to think for a while to do so is probably significant.

If you buy the first part of this philosophy, then the second part becomes really interesting. Ever listen to another programmer explain a design and wonder how the heck that was going to solve the problem at hand? Ever notice how programmers love to make up new terms for things in their code, often in defiance of the normal English-language meaning of the word? Or, even worse, ever read code where the variables had names like *info* or *data* or *value* or the functions had names like *processData*? Does anyone think names like those *really* have any informational content?

The OO philosophy suggests that the things manipulated by the program should correspond to things in the real world. They should carry the same names as those things carry in the real world. And they should interact in ways like those objects n the real world.

Design from the World

- Walk into a library and what do you see? Books, Librarians, Patrons (customers), Shelves, ... In an older library you might actually see a card catalogue!

  Of course, on the way in the door you saw the building/Branch Library.

- These things form a set of candidate ADTs. Next we would explore the very properties and relationships that these things exhibit in real life:

. . . . . .

In the OO approach to our earlier library design, we might never arrive at subsystems like *RecordStorage*, *CheckInOut*, *AccountMgmt*, and *InventoryMgmt*. Instead, we would draw upon our knowledge of the world of libraries and say that this world is populated by *Book*s, *CardCatalog*s, *Library*s, *LibraryBranch*es, *Patron*s, etc., and these would become our initial set of ADTs, our modules for the system design. If our personal knowledge of the world of libraries is not quite up to snuff, we can ask the *domain experts*, the librarians and staff and patrons of the library.

Real-World Relationships

- Books have titles, authors, ISBN, ...
- Librarians and Patrons have names. Patrons have library cards with unique identifiers. Most librarians are also patrons.

- Patrons return books. Librarians check in the returned books.
- Librarians check books out for patrons. (At some libraries, patrons can also do their own check-out.) Librarians handle the processing of newly acquired books.
- Although books can be returned to any branch, each book belongs to a specific branch and will, if necessary, be delivered there by library staff.

. . . . . .

Organizing the World

OO designers would start by organizing these things into ADTs, e.g.:

Librarian
name assignment: BranchLibrary
checkOut(Book, patron) checkIn(Book) acquisition(Book)

Book
title: string authors: seq of Author isbn: ISBN homeBranch: BranchLibrary

Branch Library
inventory: set of Book stacks: seq of Shelf staff: set of Librarian
addToInventory(Book) removeFromInventory(Book)

Catalog
author Index subject Index title Index
add (Book, BranchLibrary) search(filedName: String, value: String) : seq of CatalogEntry

. . . . . .

Simulate the World's Actions

Only then would OO designers consider the specific steps required to add a book to inventory: (update the electronic card catalog so that the book can be found, and record that the book is present in the inventory of a particular branch:

. . . . . .

Simulate the World's Actions

```
function Librarian::acquisition (Book book)
{
  BranchLibrary branch = this.assignment;
  Catalog catalog = mainLibrary.catalog;
  catalog.add (book, branch);
  branch.addToInventiory (book);
}

function Catalog::add (Book book)
{
  authorIndex.add (book);
  titleIndex.add (book);
  subjectIndex.add (book);
}
```

. . . . . .

OO Approach - Observations

Technically, does the same thing as the earlier design,but

- Division into ADTs is a natural consequence of the "observational" approach
- More concern with terminology ('acquisition" rather than 'addBook", "patron" rather than "customer", "stacks" rather "shelves"
  - Avoids programmer-isms such as BookInfo
- Separation of concerns is better (e.g., catalog functions are not built into the librarian code)

. . . . . .

What's the advantage to this?

- *We can talk to the domain experts* (librarians). Because we talk about things in the library world (not in some made-up-by-the-programmer world), their knowledge of how things really work is immediately relevant to our design. Because we use their terminology, we avoid a lot of frustration and wasted time explaining artificial and misleading programmer-ese terms:

  "OK, so the LB database..."

  "The what?"

  "The LB database - that's where we store the information that used to be in the card catalog. OK, the LB database has to store info about more than just books. You have magazines, CDs, DVDs, etc. So the LB database can contain any kind of shelvable item."

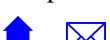
  "Shelvable item? You mean any kind of publication, right?"

  "Uh, yeah. We called them shelvable items because they represent anything you can put on a shelf. Anyway, the LB database can..."

- *We can talk to each other* on the development team. Anyone working on a software project eventually has to gain a certain knowledge of the application domain. That knowledge carries with it a whole host of expectations as to what is and is not relevant, how things behave, who does what, etc. If we know that, in the real world, librarians check out books, then when we are hunting for the code responsible for checking out books, we should feel safe in assuming it will be a function of the *Librarian* ADT, and that it will be named something like *checkOutBook*, not some bit of programmer-ese like *changeBookLocationToCustomer*.

  Grady Booch suggested the "principle of least surprise" as a guide to designing ADT interfaces. The idea is that, given that we all share a certain common understanding of how things are supposed to work, any surprises we encounter when reading someone's design are invariably unpleasant ones, representing a place where the design deviates from our expectation of how it would be most likely to work. Each such surprise is something that we are going to have to remember later, whenever we try to work with that particular designed component. If the choice of ADTs is "natural" to the application domain, if the names are natural, if the interactions (function calls) between them are natural, we have fewer surprises to cope with and fewer diddly little details likely to go wrong.

- *We can understand* our own work better. Ever go back after a couple of months, read your old code, and wonder "What in the world was I thinking?" That's just another kind of surprise, delayed. Again, if the structure of the program reflects our natural understanding of the world in which the program resides, we should avoid many of these unpleasant surprises.



Hopefully, then it's clear how an OO approach can have a big impact on the design of our programs. Closely related to OOD is OOA. If "design" is all about figuring out *how* to make a program do what we want, "analysis" is about first figuring out *what* we want it to do. But programs don't run in isolation. They interact with objects in the surrounding world. So it should not come as a shock that thinking about the world as a collection of interacting objects helps here too. If real-world librarians check out books for library patrons, then isn't it natural to assume that an automated librarian would check out books for library patrons?

2.2 Simulation == Modeling

OO Analysis & Design

- Analysis is the process of determining *what* a system should do
- Design is the process of figuring out how to do that

. . . . . .

Models

We can rephrase that as

- Analysis is the construction of a model of the world in which the program will run.
  - including how the program will interact with that world
- Design is the construction of a model of a program to work within that world.

. . . . . .

Note our approach in the earlier library examples: we built a model of the library world, and took for granted that this was the basis for design of library code.

Steps in Modeling

- Classification: discovering appropriate classes of objects
- Refined by documenting
  - interactions between objects
  - relations between classes
- Both steps often driven by scenarios

3 Putting the "Programming" in OOP

OOP in Programming Language History

Programming Languages		Design
Concepts	**Examples**	
Statements		Stepwise Refinement
Functions	FORTRAN (1957), COBOL, ALGOL, Basic (1963), C (1969), Pascal (1972)	Top-Down Design
Encapsulated modules, classes	Modula, Modula 2 (1970), Ada (1983)	Information hiding, ADTs
Inheritance, subtyping, dynamic binding (OOP)	Smalltalk (1980), C++ (1985), Java (1995)	Object-Oriented A&D

So, if the point of OOA and OOD is to come up with good ADTs, why do we need OOP? In fact, many OODs lead to programs that could be implemented very nicely in a pre-OO programming language that had good support for ADTs. But certain patterns of behavior crop up again and again in simulations, patterns that could not be easily supported in those earlier languages. The languages that we call object-oriented evolved to support these behaviors. This does not replace support for ADTs - all OOPLs start with good support for ADTs. But then they add to it the capability for class- or object-variant behaviors.

3.1 What is an Object?

Maybe you thought I was never going to get around to this?

Objects

An *object* is characterized by

- identity
- state
- behavior

. . . . . .

Identity

Identity is the property of an object that distinguishes it from all others. We commonly denote identity by

- names, such as Steve, George,
  - in programming, x, y
- references that distinguish without names: this, that,
  - in programming, *p

. . . . . .

State

The *state* of an object consists of the properties of the object, and the current values of those properties. For example, given this list of properties:

```
struct Book {
   string title;
   Author author;
   ISBN isbn;
   int edition;
   int year;
   Publisher publisher;
};
```

we might describe the state of a particular book as

```
Book cs330Text = {
   "Object Oriented Design & Patterns", horstmann,
   "0-471-74487-5", 2, 2006,
   wileyBooks};
```

. . . . . .

Behavior

Behavior is how an object acts and reacts to operations on it, in terms of

- visible state changes and
- operations on other visible objects.

. . . . . .

OK, identity, state, and behavior are nice ideas. But what's it really add up to? How do you know if you have a *good* object?

Personally, I'm rather fond of the "kick it" test. If you can kick it, it's an object. Returning again to our library example, *Book*s, *CardCatalog*s, *Library*s, *LibraryBranch*es, *Patron*s are all physical tangible objects. If I came up to one, I could give it a kick. (Some of them might kick back!) So I think I can accept all of these as (classes of) objects.

What about *RecordStorage*, *CheckInOut*, *AccountMgmt*, and *InventoryMgmt*. Have you ever walked down a hall and tripped over an InventoryMgmt? Could you kick/touch/hold a CheckInOut? That's a pretty good clue that these are not objects.

Object (and class) names are invariably noun phrases. A good rule of thumb is that tangible objects in the real world make good objects in the simulated world. This includes people. However, not all simulated objects need to be tangible. Events are often objects. Roles played by people in a system are often objects.

Some things that are terrible choices for objects early in the system development become acceptable later on. For example, is *RecordStorage* an object? It could be. It does not require a whole lot of imagination to believe that it would have identity, state, and behavior. The name isn't great – it smacks of programmer-ese. More importantly, there probably isn't one of these in the library right now, except in the form of the card catalog, in which case we are better off with a *cardCatalog* object.

But later on, when we are deep in design, if we make a design decision to unite the card catalog and circulation and inventory records into a database, calling that a *RecordStorage* object might not be quite so awful (though I'd still prefer a more descriptive name). That's because, although this thing does not exist in the old unautomated world, it will exist in the new world that includes our automated system.

3.2 Messages & Methods

Messages & Methods

In OOP, behavior is often described in terms of messages and methods.

A *message* to an object is a request for service.

A *method* is the internal means by which the object responds to that request.

. . . . . .

Differing Behavior

Different objects may respond to the same message via different methods.

Example: suppose I send a message "send flowers to my grandmother" to. . .

- a florist
- my sister
- a Department secretary
- a tree

. . . . . .

The florist responds by checking to see if my grandmother's address is local. if so, he schedules a delivery to her. If not, he consults his directory of affiliated florits to find one in the same city as my grandmother. He contacts that florits, and passes the request to them in exchange for a portion of the fee.

That is the "florist method" for responding to this message.

My sister responds to the message by a very different meother. She googles for a local florist, gets on the phone to them, and passes my request on to them.

It's a different method of response to the same message, but, either way, Grandma gets her flowers, so we're all happy.

If I send the same message to a Department secretary, she responds, "Send your own d— flowers. Who do you think I am? Your servant?". In programming terms, an exception has been thrown or a run-time error has been signaled. Now, that's not ideal – Grandma didn't get her flowers – but, still, it's something I can work with because I can detect the error message and respond to it.

Finally, if I send the same message to a tree, I get no response at all. Trees just don't accept that message.

In using objects, we must know what messages they will successfully respond to, not necessarily the method of the response. But how can we even know whether a group of objects respond to some common message? All OOPLs provide some way to organize objects according to the messages they accept. Most common is to group them by class.

3.3 What is a Class?

What is a Class?

A *class* is a named collection of potential objects.

In the languages we will study,

- all objects are instances of a class, and
- the method employed by an object in response to a message is determined by its class.
- All objects of a given class use the same method in response to similar messages.

. . . . . .

By now you might start to wonder if objects and classes are really all that new, or if I'm just playing terminology games with you. In fact, all the terms I have introduced in this lesson have direct correspondences to more traditional programming terms:

Are Objects and Classes New?

OOPL	Traditional PL
object	variable, constant
identity	label, name, address
state	value
class	type (almost)
message	function decl
method	function body
passing a message	function call

So all the new vocabulary we've been introduced to has existing equivalents in the traditional programmign world.

One advantage to the new vocabulary is that it helps get you in the mindset of thinking of the program as a simulation of actual objects.

Classes versus Types

Although "class" and "type" mean nearly the same thing, they do carry a slightly different idiomatic meaning.

- In conventional PLs, each value (object) is an instance of exactly one type.
- In OOPLs, each object (value) may be an instance of several related classes.

Although you can interchange "type" and "class" in many statements, the use of the word "type" carries with it an implication that only one is involved.

How is it possible for one object to be a member of multiple classes? It happens in the real world all the time that we divide things into multiple levels of ever-finer classes: This occurs because classes can be related via “inheritance”.

Instances of Multiple Types

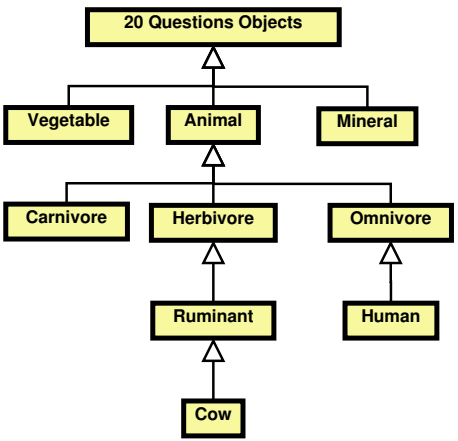
- Classes can be related via “inheritance”.
  - Inheritance captures an "is-a" or "is a specialized form of of"relationship.

If you have ever played the game "20 Questions", you may be familiar with the stereotypical opening questions:
"Is it an animal?"

If the answer is no, this is followed with "Is it a vegetable?" (meaning any plant at all). If the answer to that is false, the object is known to be a "mineral" because only tangible objects are considered fair game in most versions of 20 Questions.

If the answer to the animal question were "yes", however, this is generally followed by questions regarding its diet: "Is it a carnivore?", "Is it an herbivore?" and so on. An answer of "yes" to the herbivore question might be followed by "Does it eat grass?" (ruminants).

Inheritance Example



. . . . . .

The diagram here shows that we are progressively restricting ourselves to more specialized classes. Now, clearly Bessie the cow (a specific object) is a member fo the class *Cow* but that is nto all. She is simultaneously a member of the class *Ruminant* and of the classes *Herbivore*, *Animal*, and of the class of valid 20 Questions objects.

This ability to model some classes ans specializations of others is, together with the ability to implement variant behaviors in response to a common message, precisely what traditional PLs were unable to support but that OOPLs were designed to allow.

How do we know if a group of objects will respond to a message? If they are all members, at some level, of a class that supports that message. (Later we will look at an additional mechanism often employed for this purpose, "subtyping".)

How does an object determine what method to use in response to a message? It uses the method associated with the most specific (specialized) class to which it belongs. We will alter see that this rule is referred to as "dynamic binding".

What makes a PL an OOPL?

- It must provide support for variant behavior
  - Usually accomplished via *dynamic binding*
- It must provide a way to associate common messages with different classes.

  Usually accomplished via
  - inheritance, and
  - subtyping

. . . . . .

We'll learn what these more specialized terms mean as the semester progresses.