An Overview of the Main Course Themes

Steven J Zeil

August 20, 2013

Outline

Outline I

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.

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 sub-systems / "modules"	Too loosely defined
	gather functions & data into encapsulated ADTs	(Pre-dates OOPs by more than a decade)

Tension in Pre-OO Programming Languages (cont.)

Problem: too many. . .	**Response**	
. . . ADTs	Gather into namespaces / packages	Not much help
	Organize loosely into inheritance hierarchies	The OO approach

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?

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 `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 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.

Outline I

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.

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:

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 I

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

Organizing the World II

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)

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.

Steps in Modeling

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

Outline I

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

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 I

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

State II

```
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.

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

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

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

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

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.

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

Classes versus Types

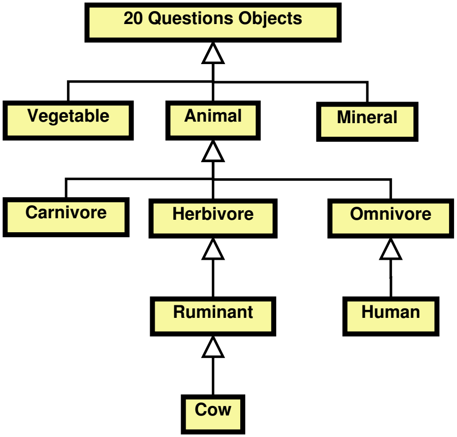
Although "class" and "type" mean nearly the same thing, they do carry a slightly different idiomatic meaning.

- 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.

Instances of Multiple Types

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

Inheritance Example



What makes a PL an OOPL?

- It must provide support for variant behavior

What makes a PL an OOPL?

- It must provide support for variant behavior
    - Usually accomplished via *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.

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

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