

Lecture 2

Modular Design
Introduction to Classes and Objects

Historical Interlude

Modular Design

Early Programming

- Assembler (at least you weren't coding in machine language)
- Basic (a "high level" language)
- Monolithic coding (think "jump" or "goto")
- Non-integrated Global Data (program-wide access)
- Problems:
 - Any code could "lobber" any other code's data very easily
 - Maintenance and New Development

Modular Programming

introduced the procedure or sub-routine

- Modular Programming facilitated a *structured* approach to Programming
 - An attempt to group code and data together into logical functions
 - But with global data, you can always “trash” some other code section’s global data unintentionally
- Global and Scoped Data (global, file scope, automatic “function” scope)

Modular Programming

introduced the procedure or sub-routine

- Example Languages:
 - COBOL (module – but everything is global within the module)
 - Experience of incrementing a counter in one paragraph, forgetting that you're also incrementing it in another
 - Data passed from one module to another is always passed by reference via the linkage section and is always global
 - FORTRAN (subroutine and function)
 - C (function)
 - Has the notion of “file scope” and “automatic” variables
 - Others: RPG, PL/1, Ada, Pascal, Haskell

Modular Programming

- Some advantages of modular programming (from Etter, *Structured Fortran 77*, p. 222):
 1. You can write and test one module separately from the rest of the program
 2. Debugging is easier because you are working with smaller sections of the program
 3. Modules can be used in other programs without rewriting or retesting
 4. Programs are more readable, thus more easily understood, because of the modular structure
 5. A module can be used several times by the same program

Modularization Revisited

- Dual motivation:
 - Reuse
 - Functionality localized into a procedure could readily be reused through the concept of a function or procedure call
 - Protection
 - Data localized into one place was protected from global access (and therefore the introduction of bugs due to inadvertent access)
 - Functionality localized into one place was protected in the sense that if that code needed to be changed, it only needed changing in one place, not in several disparate places

Modularization Revisited

- In the end, modularization helps control complexity by hiding the details
 - Example: Telephone and its interface
 - Example: Modular office/furniture
- In the object landscape, the notion of functional modularization is enhanced into a data and functional *encapsulation*
- In OO, Classes thus, in the words of Meyers, “play two roles which pre-O-O approaches had always treated as separate: module and type”
- A Class is thus both a module and a type, at the same time

Evolution of General Languages

- First Generation
 - Fortran I, Algol 58
- Second Generation
 - Fortran II, Algol 60, COBOL, Lisp
- Third Generation
 - PL/1 (based on Fortran, Algol, and COBOL)
 - Algol 68
 - Pascal
 - Simula
- Fourth Generation (Forth, etc., then the deluge...)
- But the key question: How do we protect data?

Fortran Simulation Example

- See sample code of a single-server queue simulation in [Fortran](#) and C
 - What features do you see?
- Modularization organized *code*, but still did not solve the problem of global data, nor did it allow for the conceptualization of encapsulated data-function pairs
- Modularization still kept separate the programmatic concerns of the *relationship* between *state* and *behavior*
- Modularization still did not allow for the creation of *new* non-primitive types that encapsulated *both* state and behavior

Simulation Models and Mimesis

- Modeling
 - visual abstraction
 - communication motivation
- Simulation of Discrete Dynamic Systems
 - Simula ('60's)
 - class, inheritance, polymorphism
 - the goal was simulation, classification was the method
 - naming entities *from* the problem domain itself
- Taxonomy and Language
 - Type classification and discovery
 - classes and instances

Simula

- Simulation of Discrete Dynamic Event Systems by Nygaard and Dahl at the Norwegian Computing Center (NCC) in Oslo between 1962 and 1967—Introduced the following concepts:
 - Class, inheritance (single only), polymorphism
 - Garbage collection and the *new* keyword
 - Naming entities *from* the problem domain itself
 - They wanted to have a general purpose programming language that would at once be the system *description* (specification) as well as the system *prescription* (implementation) (Cf. Alan Kay on Lisp)
 - Abstract Data Types
- Simula Code Example

Alan Kay on Lisp

experience with both of these new technologies was critical for the more radical systems to come.

One little incident of LISP beauty happened when Allen Newell visited PARC with his theory of hierarchical thinking and was challenged to prove it. He was given a programming problem to solve while the protocol was collected. The problem was: given a list of items, produce a list consisting of all of the odd indexed items followed by all of the even indexed items. Newell's internal programming language resembled IPL-V in which pointers are manipulated explicitly, and he got into quite a struggle to do the program. In 2 seconds I wrote down:

```
oddsEvens(x) = append(odds(x), evens(x))
```

the statement of the problem in Landin's LISP syntax—and also the first part of the solution. Then a few seconds later:

```
where odds(x) = if null(x) v null(tl(x)) then x  
                  else hd(x) & odds(tl(x))  
evens(x) = if null(x) v null(tl(x)) then nil  
                  else odds(tl(x))
```

This characteristic of writing down many solutions in declarative form and have them also be the programs is part of the appeal and beauty of this kind of language. Watching a famous guy much smarter than I struggle for more than 30 minutes to not quite solve the problem his way (there was a bug) made quite an impression. It brought home to me once again that "point of view is worth 80 IQ points". I wasn't smarter but I had a much better internal thinking tool to amplify my abilities. This incident and others like it made paramount that any tool for children should have great thinking patterns and deep beauty "built-in".

Simula's Power to the People

It is a common belief here that this Conference consists of people belonging to the 1% group: this is not true, since I am a representative of the 99% group and I think since we are so many and we are going to use these tools, we should have the right to say something here. We are simple-minded men but we have complex problems and we are forced to face these complex problems. [...]

When [Jan] Garwick talked about his language [GPL], he said this was a super language which solved all problems. Of course, even if it looked a little frightening still I was relieved. But then what happened? He said that this is just for the 1% group but for the 99% we are going to provide a number of different languages which are defined in the general language so, therefore, we are forced to learn all these languages instead. This is not what we want. What we want is to be given concepts so that we can handle complex problems in a manner which we are able to grasp.—Kristen Nygaard, Lysebu Conference on Simulation Programming Languages in May of 1967, quoted from Jan Rune Holmevik, *The History of Simula*, 1995

Benefits of Object-Oriented Simulation

1. It promotes reusability because existing objects can be reused or easily modified
2. It helps manage complexity by breaking the system into different objects
3. It makes model changes easier when a parent object can be modified and its children objects realize the modifications
4. It facilitates large projects with several programmers

—Law & Kelton, *Simulation Modeling and Analysis*,
3rd. ed.

Traditional Decomposition

- Early computer programs (1950' s and 1960' s) were implementations of algorithms used to do fairly basic activities, such as aeronautical trajectory, beam analysis, fuel analysis, earthquake analysis, Gaussian density, etc.
- These programs tended to be primarily algorithmic-centric or data-centric
- The languages used to implement these programs were *machine-derived* and expressed a low-level vision of the world

Traditional Decomposition

- As hardware capacity improved (1970' s on), the viewpoint became abstracted higher into larger visions of *applications* (rather than *programs*), focusing on larger-scale systems such as business applications (policy management systems in Insurance, inventory management systems, etc.)
- More and more capabilities were being demanded by the user communities, with a new focus now on *systems* rather than *programs*

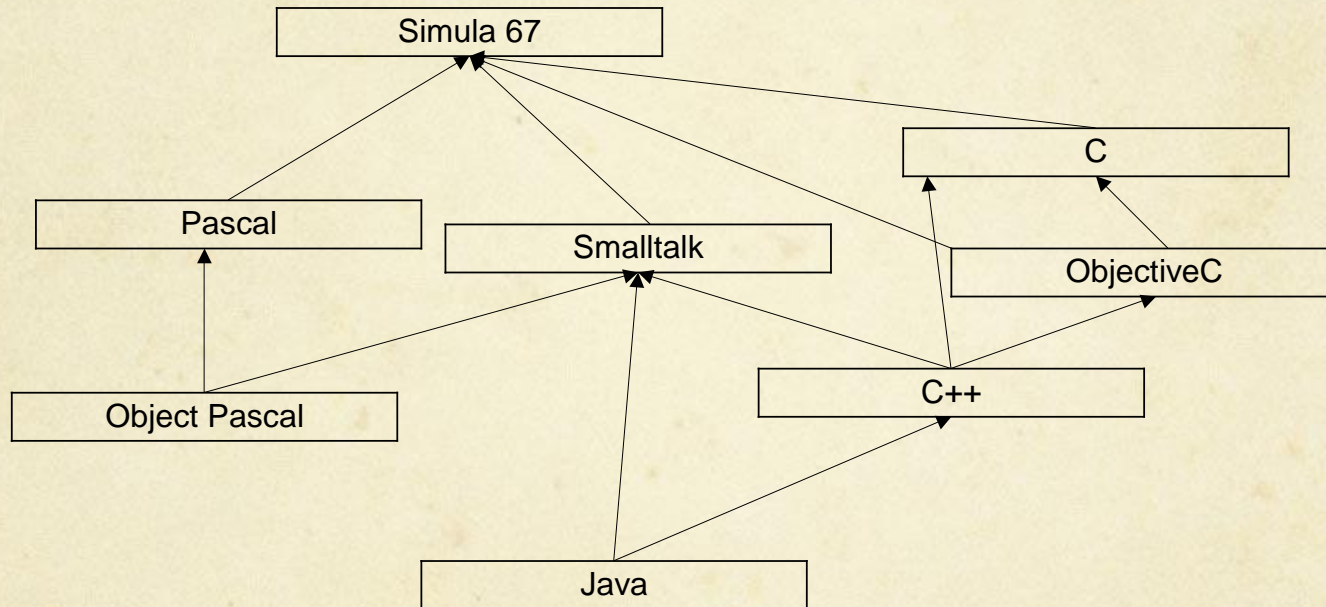
The Problem

- The problem was that the same programming languages used to model the smaller-scale programs were now being utilized to solve the much more abstract, demanding, and significantly more complex applications
- Programmers were still writing code with floating point variables, integer variables, having to worry about linking segmentation, and still forced by the procedural languages themselves to worry about hardware issues (program segmentation, etc.)
- While *at the same time* having to keep an eye on the ever-increasing complex requirements that were coming in from the user communities

The Solution

- The solution that was proposed in Simula was to unite the algorithmic and data-centric views of procedural applications, and begin to work with larger-scale types rather than simply integers and floats
- This allows programmers to begin to think and speak in the *language of the users* rather than in the *language of the machine*
- Now they could talk about policies, contacts, customers, traffic jams, airplanes, i.e., real world entities
- Thus, OO programming languages bridge the semantic gap between the *users* and the *machine*
- They did this through the introduction of the concept of a *Class*

Partial History of Object Oriented Languages



Introduction to Classification

Aristotelian Foundation of Classification: The “ycleptic Factor”

- ὁμώνυμα λέγεται
 - equivocally named (homo = "same" in terms of *name*)
- Example: a plumbing “pipe” and a smoking “pipe” (in a portrait) (cf. Magritte)
 - contextual predication (cf. equivocation/amphiboly fallacy in Aristotelian Logic)
 - non-essential—Identification in *name* only
 - -> homonyms: ”stalk" and ”stalk", ”cell" and ”cell”
- συνώνυμα λέγεται
 - unequivocally named (syn = "same" in terms of *type*)
 - Example: “man” and “ox” are both "animals" (*Classification*)
 - essential identification in terms of *common class*
 - -> synonyms: "big" and "large“, both *types of sizes*

Taxonomy and Language: Context and Reference

- Linnaeus (1737): replaced a traditional abstract top-down taxonomy with an *a posteriori*, inductive approach, starting with common species and then categorizing them
 - Central question: what do these *individuals* have in *common*?
 - This *point d'appui* of Linnaeus' s is important from a system design standpoint—how do we go about identifying entities in a problem domain?

So What's a Class, Anyway?

- A class is a type and as such is a partial or full implementation of an Abstract Data Type
- A class is a general description of objects *from a specific problem domain* that share identical semantics and attributes
- A class is a collection of related subroutines packaged together, along with a definition of the data that the subroutines manipulate or use (Cockburn paraphrased)
- A class is a software construct describing an abstraction drawn from a problem domain and optionally, its implementation.

Classes Concepts

- In OO, a Class is the central unit of modularization
- Every class defines a new *type*. A class may or may not define an implementation of its semantics (provide a full implementation of its methods). A class which does not provide a full implementation is called an *Abstract Class*.
- In OO, classification is the process of *creating a new language* drawn from a particular problem domain.
- A class is an ADT that defines behavior in terms of methods and state in terms of attributes

Classes Concepts

- A Class *conceptualizes* the data and function necessary to implement a subset of user requirements
- A Class is “the descriptor for a set of objects that share the same attributes, operations, relationships, and behavior.” —Rumbaugh
- A Class is a set of objects that share a common structure and a common behavior (protocol).—Booch
- A well-designed Class provides a *crisp abstraction* of some thing drawn from the *vocabulary* of the problem domain or the solution domain.—Booch

... from a specific Problem Domain

- **Insurance:** Claim, Policy, Bill, Provider, Coverage, Obligation
- **Accounting:** Invoice, Payment, Interest, Check, AutomatedPayment
- **Banking:** Customer, Account, CheckingAccount, SavingsAccount
- **Air Traffic Control System:** Plane, Airport, Alarm
- **University Registration System:** Student, Instructor, Course, Section

State and Behavior

- Behavior is encapsulated in the code necessary to the *conduct* the fulfillment of some set of requirements defined for a class of things
- State is the *data* that are necessary for the individualization of the class instance in fulfilling its requirements
- A class satisfies the requirements of a given abstraction defined in a given problem domain

Behavior Fundamentals

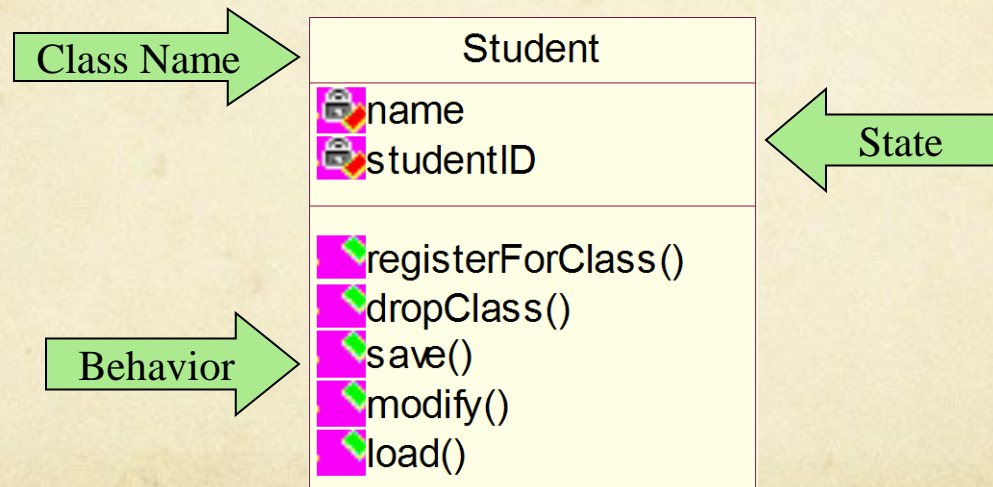
- Behavior is implemented by defining operations on a class of things
- Behavior defines the implementation for the requirements of a class
- Instances of a Class can be *treated* in the same way:
 - All Policies can be *renewed* and *cancelled*
 - All Accounts can be *opened* and *closed*
 - All Checks can be *issued* and *deposited*
 - All Payments can be *processed* and *credited*
 - All Planes can *descend* and *land*
 - All Trades can be *executed* and *validated*

State Fundamentals

- State is implemented by defining Attributes on a class of things (and also relationships—but later)
- State is the place in a Class definition where the individualization of a particular instance of a class is stored
 - An Account's *balance* and *number*
 - A Trade's *execution price* and *date*
 - A Plane's *altitude* and *speed*
 - A Policy's *premium* and *beneficiary*
 - A Course's *number of students enrolled*

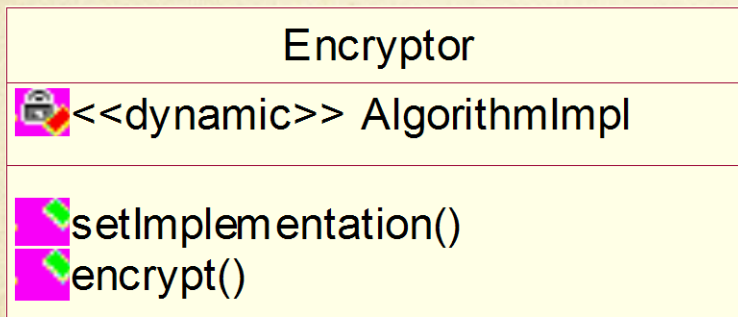
What is defined for a Class?

- State (encapsulated as *Attributes* in UML)
- Behavior (encapsulated as *Operations* in UML)



Methods

- There may be various *ways* in which an operation may be implemented.
- These various ways are the *methods by which* a given requirement may be implemented
- These methods may be encapsulated as implementations of a common operation defined in the Class' s interface



There may be various *methods* by which the *encrypt* operation may be implemented (DES, RSA, Blowfish, etc)

Synonyms I

- The Class' s *Operations* are sometimes called:
 - Protocol
 - Behavior
 - Interface
 - API
 - Services
 - Methods
 - Member functions

Synonyms II

- The Class' s *Attributes* are sometimes called:
 - State
 - Member variables
 - Instance variables
 - Class variables (Static variables)

Class Attributes

- Attributes that belong to a class of things are called Class Attributes
- Often called “static” as in static variable and static method
- A Class Attribute is shared among all runtime instances of a given class
- Examples:
 - Count of items on a Stack
 - Totals

Class Methods

- A Class Method (or static function) is a method defined for a class that belongs to the class and *not* to an individual instance
- This means that no object needs to be present in order for the method to be called—the method is called *on the class*
- There is no inherent *this* pointer associated with a class method (therefore class methods can only access class variables)
- Class methods cannot be called polymorphically
- Class methods are used primarily as encapsulated methods for Class attributes

Member Visibility

- Classes may have certain protection mechanisms that determine the degree of *visibility* any given attribute or operation may have:
 - public
 - private
 - protected
 - package

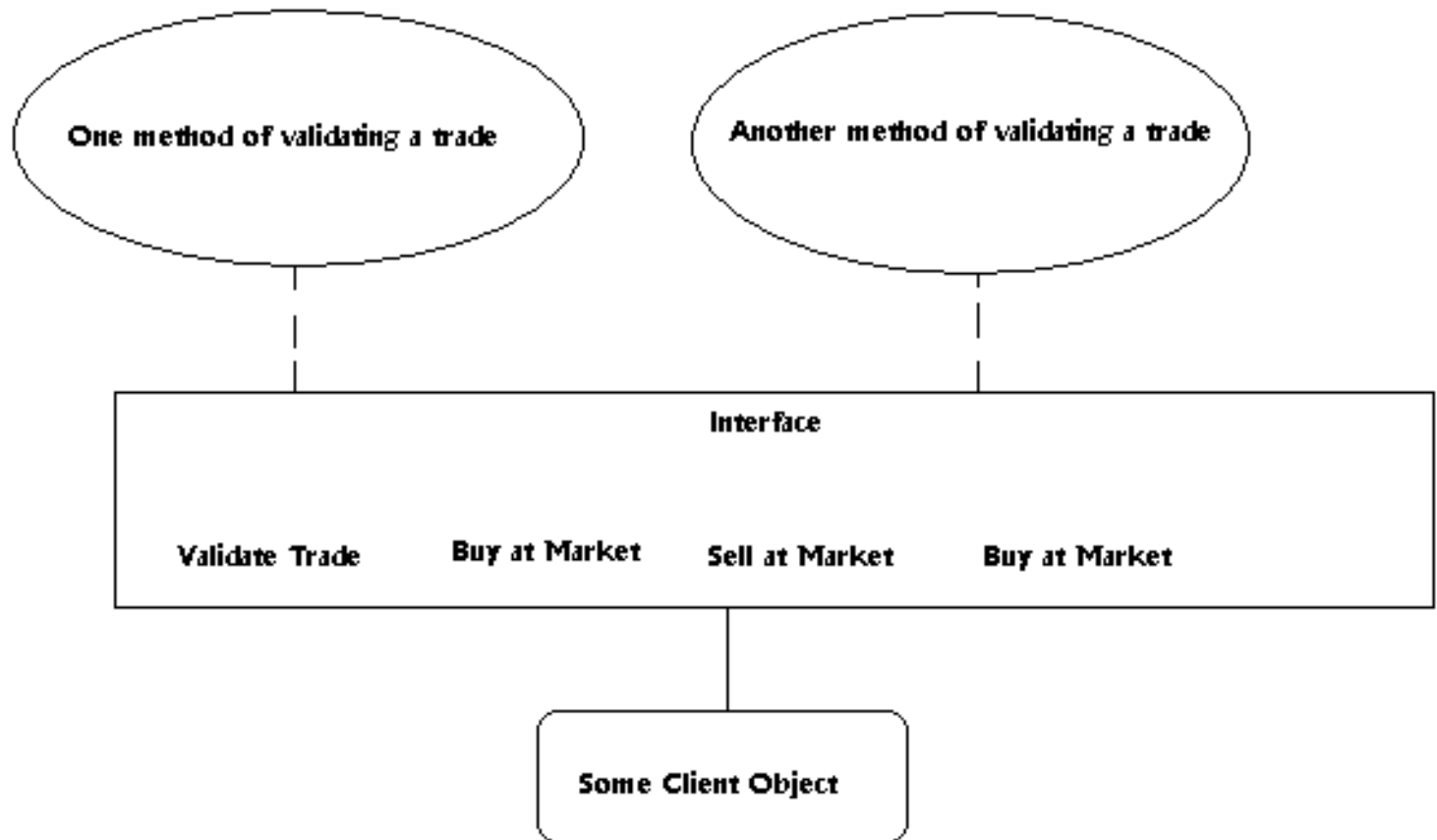
Class Comparisons

- See [handout](#) for the same simple Geometry classes written in:
 - Simula
 - Java
 - C++
 - Smalltalk
 - CLOS (Common Lisp Object System)

Interface and Implementation

- Operations defined for a Class represent that Class' s *interface* (sc. to user' s of the Class)
 - It' s the *language* other Classes use to communicate with the Class
- For any given operation, code may be defined as the *method by which* a requirement is accomplished
- In some languages, an operation is called a *method* or a *function*
- Always remember that an operation is one method by which a requirement is accomplished, among other methods
 - “What method do you use to ...?”

Interface Abstraction



Encapsulation Revisited

- Modular motivation: Hiding the implementation from the client
 - Less dependence on particular implementations
 - fewer assumptions by the client
 - clients can only manipulate encapsulated entities, not the internals
 - since clients can only interact through defined interfaces (methods), the implementation can change without affecting the clients
- Encapsulation provides limited and controlled access to an object's state, through public, protected, and private methods.

So What's an Object?

- A run-time named instance of a Class with a unique identity
- An object is not a Class. Do not confuse the mold with the product, the blueprint with the house
- An object has *individual* state, behavior, and identity. (Booch)
- An object is an instance of a Class, similar to the way a variable is an instance of a data type in a programming language
- An object exists only in computer memory, a Class exists in your mind and in your design

Mommy, Where Do Objects Come From?

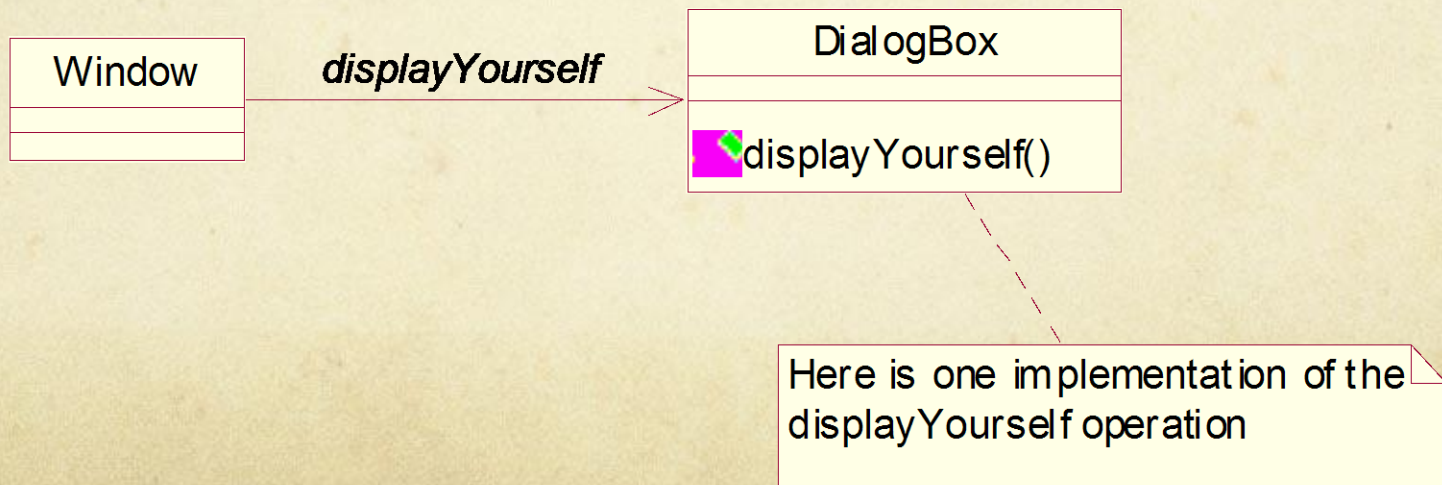
- If Classes, as Kant says, come from your brain, where do Objects come from?
 - Classes as blueprints or molds (C++ motivation)
 - Creepy Crawlers ThingMaker by Mattel
 - Molds
 - Creepy Crawlers
 - Constructors
 - Destructors (in non-garbage-collected languages)
 - Classes as factories (Smalltalk motivation)
- In most OO languages, we create new objects using the new keyword

Classes and Instances

- An object is an instance of a class when it has a name and an address (reference).
- Examples:
 - Class: Flight Instance (object): TWA 2345 on 08/15/2010
 - Class: Employee Instance (object): Bob Smith, SSN 123-45-6789
 - Class: Date Instance (object): 08/15/2011
 - Class: Account Instance (object): Wilma Meyers, balance \$123.80

Object-Object Communication

- Objects communicate by *sending messages* to other objects
- These messages are part of the defined interface of the other class
- An object must have an implementation of every operation defined in its interface



Factories

- Factories are objects that create other objects on request
- Factories can hide the class of an object from the requester, delivering a generic type back which the client can then use polymorphically and ignorantly of the actual class (eg: A “Window” object with varying look and feel)
- Sometimes you don’ t want the client choosing the type of object to create—you want to hide *new*
- By encapsulating object creation in a factory, when a new type is added, only the factory needs to be modified

Metaclasses

(or, Kant was only partially correct...)

- Metaclasses are the classes of classes
 - Every Class has a Metaclass (Smalltalk)
 - Each Class is the only instance of its Metaclass
 - Smalltalk: Metaclass allInstances size. [*print it*]
 - Smalltalk: CSPP51023.Rectangle superclass. [*print it*]
- Metaclasses encapsulate information about a Class, they *objectify* classes, rendering Classes as *objects of interest*.
- Primary uses:
 - Object creation
 - Introspection (Reflection)
 - Omnipotence ;-)

Features of the “Object Model”

- Classes (module and type)
- Inheritance and Polymorphism
- Declarative Interfaces and Deferred Implementations
- Event Handling
- Message Passing
- Object Lifetime (“From new to delete”)
- Exception Mechanism
- Garbage Collection

Complexity Busters?

- Abstraction
 - Classification
 - Conceptual Encapsulation
 - Separate Behavior (Semantics) from Implementation
- Object Orientation
 - Hierarchical Imposition (Inheritance classification)
 - Encapsulation and Modularity
 - Polymorphism
 - Process Offerings: Incremental, Iterative, etc.
 - Promise of Reuse