The University of Iowa
CS:2820 (22C:22)
Object-Oriented Software Development
Spring 2015

The Object Model
by
Cesare Tinelli
The *Object Model of Development*

- Built on the best ideas from previous technologies
- Influenced by major trends in software engineering:
  1. increased focus on programming-in-the-large
  2. evolution of high-level programming languages
Object in Object-Oriented Programming Languages

Entity that

• combines features of
  • procedures: performs computations
  • data: stores local state
• is characterized by certain invariants
Essence of
OO Programming

• Programs are organized as cooperative collections of objects

• Each object is an instance of some class

• Classes are related via an inheritance relationship
• **Builds a model** of the real-world using an object-oriented view

• **Examines requirements** in terms of classes and objects found in the problem domain
OO Design

- Leads to an **object-oriented decomposition**
- Uses various notations (e.g., UML diagrams) to express **various views** of the system being designed:
  - **logical** (classes and objects) vs. **physical structure** (modules and processes)
  - **static vs. dynamic aspects**
OO Software Development

- The products of OO Analysis serve as starting points for OO Design

- The products of OO Design serve as blueprints for an OO implementation
The Object Model of Development

Is built on the synergy among:

• abstraction
• encapsulation
• modularity
• hierarchy

• typing
• concurrency
• persistence
Abstraction

• The **process** of identifying similarities between objects, situations or processes and ignoring their differences

• A **description**, or specification, of something that emphasizes some details or properties while ignoring others

• It focuses on the **essential characteristics** of something **relative to a viewer's** perspective
Abstraction

• **Main trait:** it can be understood and analyzed independently on how it is realized

• **Quality:** it is relative to its viewers/users and their current needs

*Establishing the right set of abstractions for a problem domain is the main challenge of design*
Abstraction in OO Design

- We can characterize the **behavior** of an object, the **server**, in terms of the **services** it provides to other objects, the **clients**.
Abstraction in OO Design

• An object's abstraction defines a **contract** that
  • other objects depend on and
  • must be honored by the object

• The contract establishes all assumptions a client may make about the behavior of the server
Design by Contract

• Each service (operation) provided by an object has a set of

  • **preconditions**, to be satisfied by the client when invoking the service

  • **postconditions**, guaranteed by the server upon completion of the service

  • **invariants**, properties maintained between operations
Abstraction Examples

• Temperature sensor

• Point on a grid

• Bank account
The Object Model of Development

Is built on the synergy among:

✓ abstraction
• encapsulation
• modularity
• hierarchy

• typing
• concurrency
• persistence
Encapsulation

• The **abstraction** of an object should **precede** any decisions about its implementation

• **Implementation** details should **not** be accessible to clients

• **Encapsulation** is the process of **hiding** such details
Encapsulation

- **Achieved** in OO languages by hiding the **internals** of an object (attributes and method implementations)

- **It greatly facilitates changes** that do not impact the abstraction (i.e., the object's contract)

- Leads to a clear **separation of concerns** (contract vs way to honor it)

- **Localizes** design decisions likely to change
Encapsulation in OO Languages

Classes of objects described in two parts:

- **interface**
  captures outside view of the object and its essential behavior

- **implementation**
  provides a representation of the abstraction and the mechanisms to achieve its behavior
Encapsulation Examples

- Heater Controller
- Point on plane
- Queue
The Object Model of Development

Is built on the synergy among:

✓ abstraction
✓ encapsulation
• modularity
• hierarchy

• typing
• concurrency
• persistence
Modularity

• Modularization divides a software systems into components, **modules**

• Modules
  ▶ may have **connections** to other modules
  ▶ but can be **compiled separately**
  ▶ **encapsulate** sets of **classes** and **objects**
  ▶ have an **interface** and an **implementation**
Crucial Point

- Classes and objects define a system's logical structure
- Modules define a system's physical structure
- The two structures are by and large orthogonal
Module Decomposition

• Decomposing a system into modules presents challenging design decisions

• There is a tension between the desire to encapsulate abstractions vs need to expose some of them to other modules

• General approach:
  - group together logically related classes and objects and
  - expose only those that are strictly necessary to other modules
Modularity

• **Desiderata** of module decomposition:
  - Modules
    - designed and implemented independently
    - simple enough to be fully understandable
  - Ability to change a module's implementation without
    - knowing that of other modules
    - affecting their behavior
  - Reuse
The Object Model of Development

Is built on the synergy among:

✓ abstraction
✓ encapsulation
✓ modularity
• hierarchy

• typing
• concurrency
• persistence
Hierarchy

• A (partial) ordering of abstractions

• Most important hierarchies
  ▸ "is a" relation (class structure)
  ▸ "part of" relation (object structure)
Class Structure

• The "is a" relation we consider is one that relates classes

• Examples

- A dog is a mammal
- A dog is a pet
- Fido is a dog (Fido is not a class)
When a B is an A we also say that

- **B is a subclass** of A:
  - every instance of B is an instance of A
- **B extends (or specializes)** A:
  - B has all features and behaviors of A, and possibly more
- **B inherits from** A:
  - B inherits A's features and behaviors
Class Structure

When a B is an A we also say, symmetrically, that

- A is a superclass of B:
  - every instance of B is an instance of A

- A is extended by (or generalizes) B:
  - B has all features and behaviors of A, and possibly more
Inheritance Hierarchies

• **Single** inheritance:
  ▸ each class extends (inherits from) at most one class
  ▸ the hierarchy is a tree, or a forest

• **Multiple** inheritance:
  ▸ each class extends one or more classes
  ▸ the hierarchy is graph
Inheritance

Single inheritance

Multiple inheritance
The Object Model of Development

Is built on the synergy among:

✓ abstraction
✓ encapsulation
✓ modularity
✓ hierarchy

• typing
• concurrency
• persistence
Typing in Programming Languages

• A **type** is a **collection of values with same structural or behavioral properties**
  - Ex: integer, string, integer list, integer array, integer and string pair, ...

• The **type system** of a language
  • imposes a **division** of values into types
  • defines **typing restrictions** for each operation (allowed input types, resulting output type)
Types in Programming Languages

• A language is **typed** if it **enforces** a **type system**

• It is **untyped** otherwise, that is, if it allows operations to be applicable to any values

• Note:
  • Most **highly-level languages** are **typed** to some degree (strongly/weakly typed)
  • All **assembly languages** are **untyped**
Types in OO Programming Languages

• Every class defines a type, consisting of all objects that are instances of that class

• However, not all types are classes. E.g.:
  ▸ Java's basic types (int, bool, ...)
  ▸ Java's interfaces
  ▸ Traits in Scala
Static vs Dynamic Typing

- **Statically typed languages** enforce typing restrictions at **compile time**:
  - the **type** of each expression denoting a value is determined and checked before running the program

- **Dynamically typed languages** enforce typing restrictions at **run time**:
  - types are determined and checked as expressions are evaluated
Static vs Dynamic Typing

• In **statically typed** languages **types** are associated to expressions in the source code
  ▶ C++, Java, Scala, ML, Haskell, ...

• In **dynamically typed** languages **types** are associated to values in memory
  ▶ Python, Ruby, Perl, Javascript, ...
Enhanced Type Systems

- **Overloading**: same name for different operations
  - E.g.: + for integer addition, string concatenation, list append in Scala

- **Subtypes**: types extending others
  - E.g.: subclassing in OO languages

- **Subtype polymorphism**: same name for inherited operations
  - E.g.: inherited methods in OO languages
Enhanced Type Systems

• **Parametric types**: structured types with components of arbitrary type
  - E.g.: List[X], Array[X], List[(X,Y)] for any types X, Y in Scala

• **Parametric polymorphism**: generic operations for parametric types
  - E.g.: reverse: (l:List[X]) List[X], head:(l:List[X]) X in Scala