Design Strategies in OO Development

- Abstraction: modeling essential properties
- Separation: treat what and how independently
- Composition: building complex structures from simpler ones
- Generalization: identifying common elements

Abstraction Modeling entities in software

- Only essential aspects should be captured
  - attributes
  - behavior

Practical Abstraction

A named collection of attributes and behavior relevant to modeling a given entity for some particular purpose.

Desirable Properties:

- well named: name conveys aspects of the abstraction
- coherent: makes sense
- accurate: contains only attributes modeled entity contains
- minimal: contains only attributes needed for the purpose
- complete: contains all attributes and behavior needed for the purpose
Better Abstractions

In programming, the independent specification of an interface and one or more implementations of that interface.

What is to be done

How it is to be done

Mapping Abstraction to Software

real-world

abstraction

software

entity

attributes

{data, data,…}

behavior

{method, method,…}

Separation of Interface from Implementation

In programming, the independent specification of an interface and one or more implementations of that interface.

Interface

Implementation

Interchangeability of Implementations

Allows the creation of multiple implementations with a common interface.

For example: a List ADT could use a dynamic linked list or a dynamic array for the underlying physical data structure. In either case, the same interface would be appropriate (and the user need not be concerned with the underlying structure in many cases).

Implementations that share a common interface are said to be “plug compatible”.

They may differ in algorithmic complexity, reliability, platform dependencies, etc.
Specificity of Interface

Also allows a single implementation to support multiple interfaces.

This allows the isolation of restricted set used in one situation versus another.

For example, we could have a very general List ADT that supported both standard List operations, and also Stack operations. By "subsetting" the functionality of the ADT into separate interfaces, we could provide both categories of operation, in a natural way, without duplication of shared code.

In essence, we view the implementation as a library of related widgets.

Mapping Abstraction to Software in OO

real-world  |  abstraction  |  OO software

attributes  |  {data, data,…}  |  {method, method,…}

entity  |  behavior

General Structure of a Class

class: a named software representation for an abstraction that separates the implementation of the representation from the interface of the representation

A class models an abstraction, which models an entity (possibly "real").
A class represents all members of a group of objects ("instances" of the class).
A class provides a public interface and a private implementation.
The hiding of the data and "algorithm" from the user is important. Access restrictions prevent idle or malicious alterations.
Separation and Classes

If we have two different classes, objects of each can see only the (public) interfaces of the objects of the other.

**class A**
- implementation provides methods
- class A interface identifies available methods

**class B**
- implementation can use class A methods identified in the class A interface

Multiple Instances of a Class

Each instance, or object, usually has different values for the class-defined properties.

Class = Factory   Objects = Products

Multiple Instances of a Class

SalesPerson
- private Name
  - commissionRate
  - totalSales
- public
  - sellCar
  - reportSales

Joe Hokie
- 16% $250,000 sellCar reportSales

Jill Hokie
- 16% $275,000 sellCar reportSales

When developing abstractions, or classes, it may help to think of them as people-like entities with responsibilities and collaborators.

- Responsibilities of knowing (respond with information to a query)
- Responsibilities of doing (act on something, transform, move, sort, etc.)

Collaborators: associated objects in the system with their own responsibilities

Software Engineering Goals

Objects and classes help programmers achieve a primary software-engineering goal: reusability

- A single class is used repeatedly to create multiple object instances.
- More importantly, encapsulation prevents other developers from inadvertently modifying an object’s data.

Separation allows different implementations to be used for an interface.