

Progression of Roles

Design Strategies in OO Programming

Abstraction

Practical Abstraction

Abstraction

Better Abstractions

Mapping Abstraction to Software

Separation of Interface from Implementation

Interchangeability of Implementations

Specificity of Interface

Mapping Abstraction to Software in OO

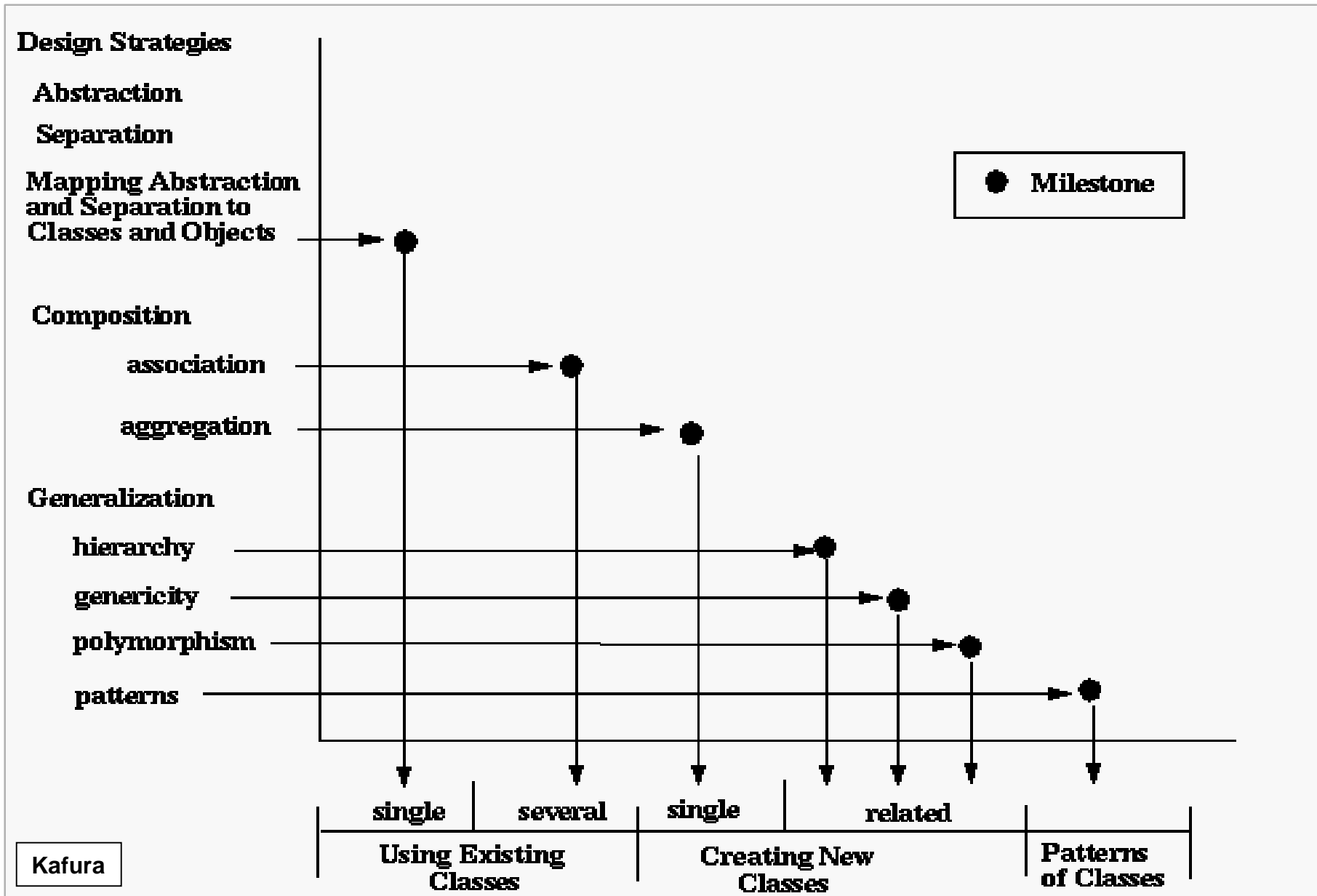
General Structure of a Class

General Structure of an Object

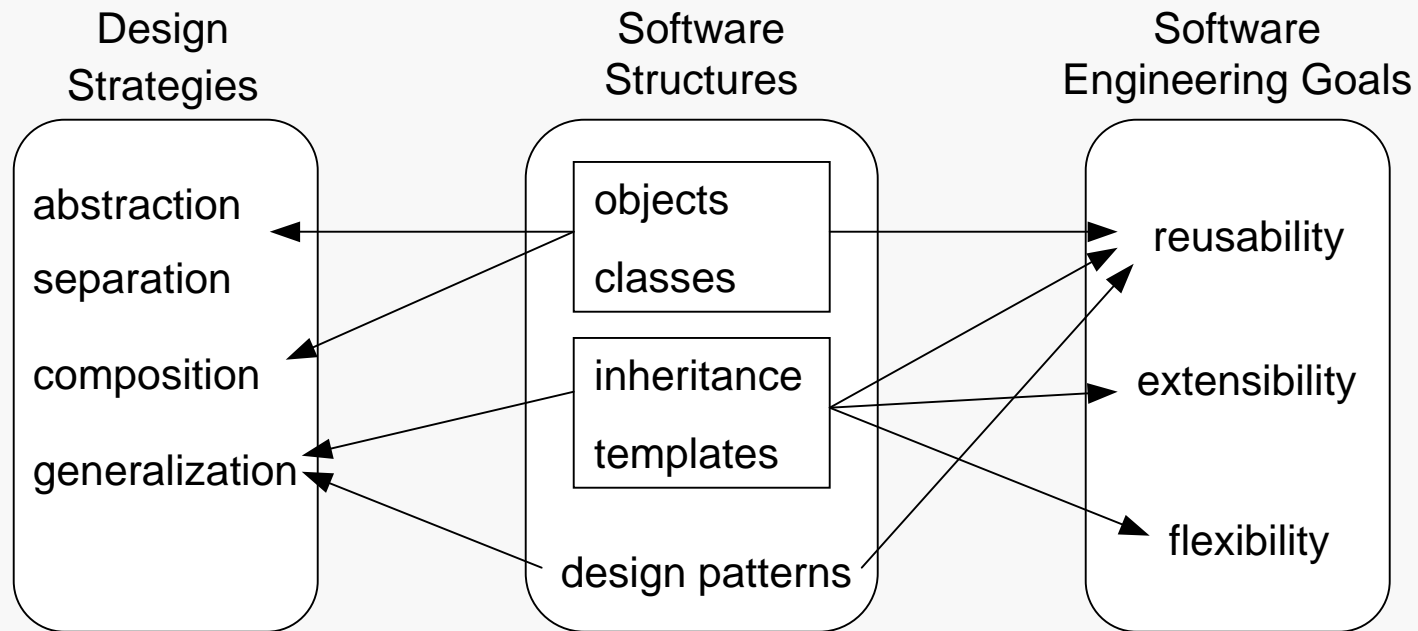
Multiple Instances of a Class

Software Engineering Goals

Progression of Roles



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



Modeling entities in software

Only essential aspects should be captured

- attributes
- behavior



Wassily Kandinski
Cossacks, 1910-11



Critique??

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

Desirable Properties:

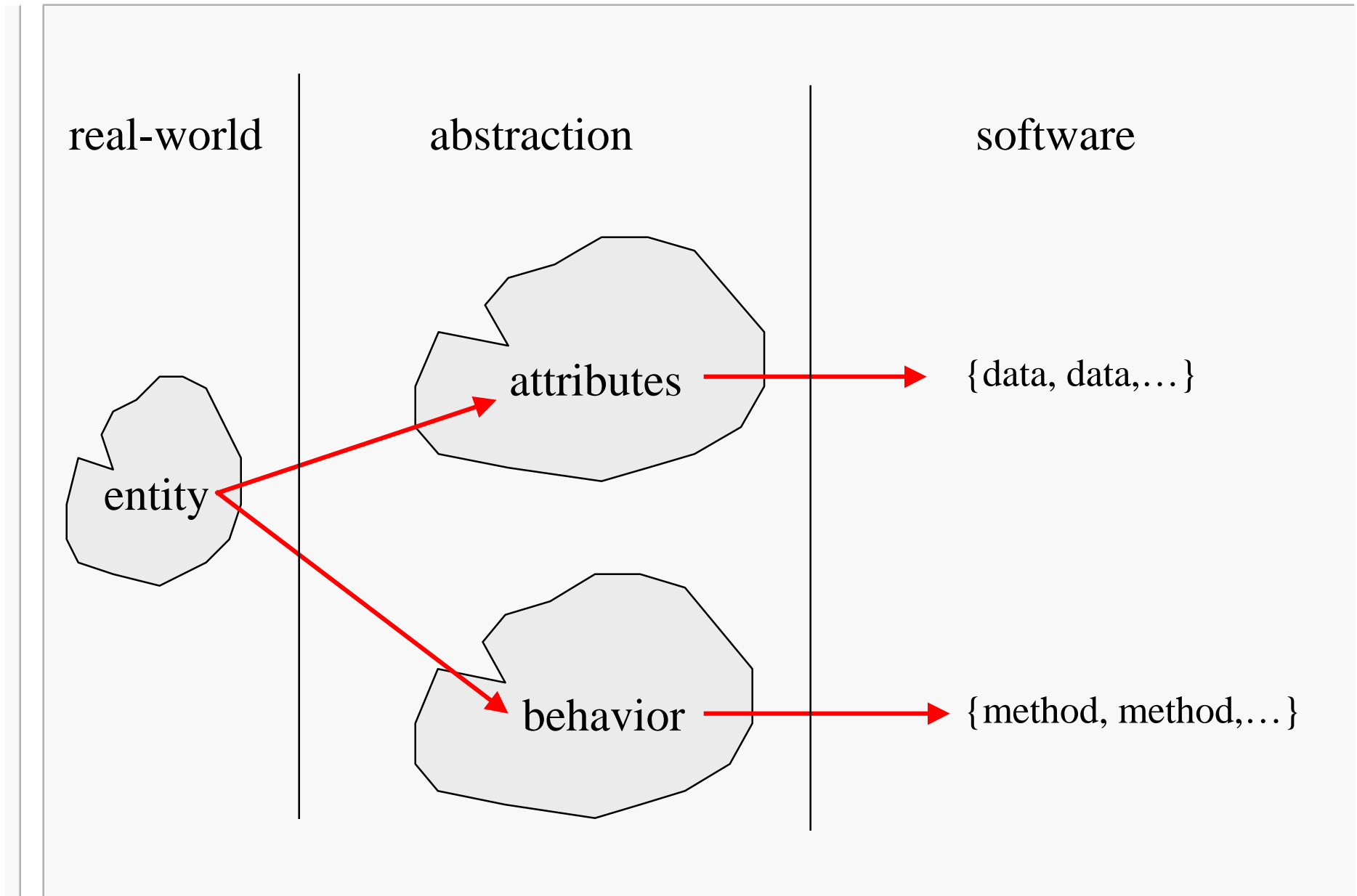
well named	name conveys aspects of the abstraction
coherent	makes sense
accurate	contains only attributes modeled entity contains
minimal	contains only attributes needed for the purpose
complete	contains all attributes and behavior needed for the purpose



MEN



WOMEN

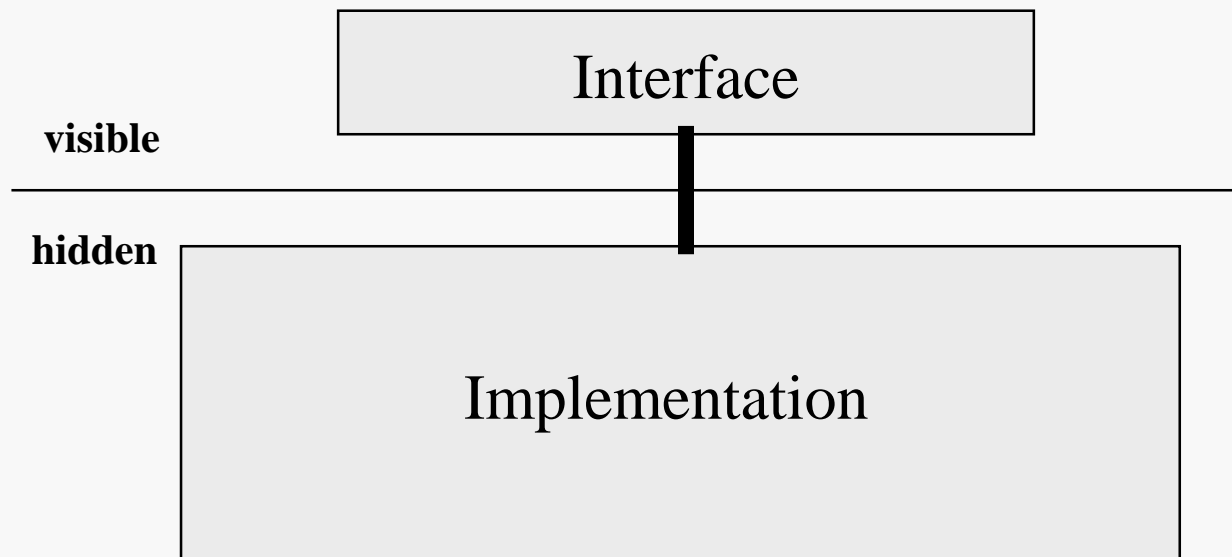


In programming, the independent specification of an interface and one or more implementations of that interface.

What is to be done

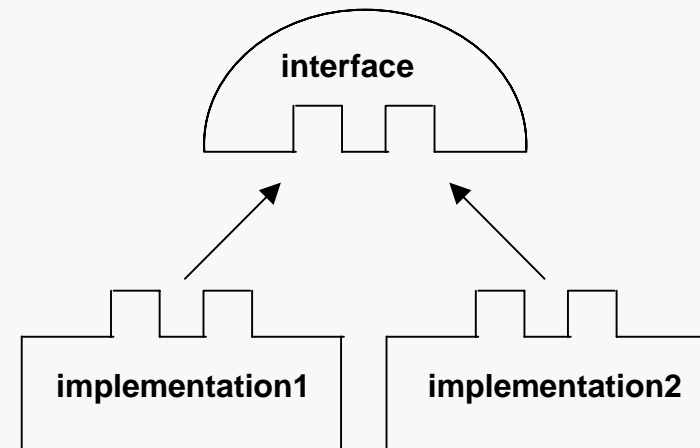
vs

How it is to be done



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).

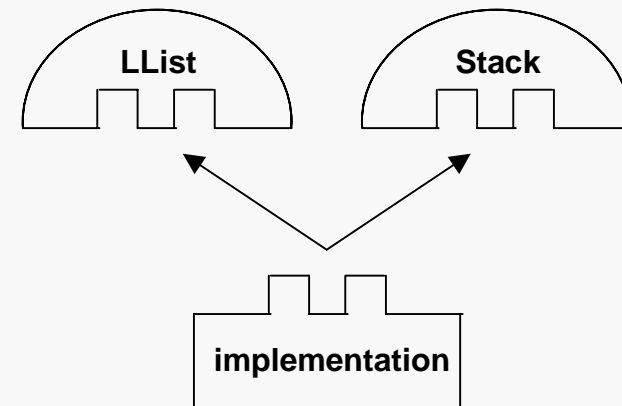


Implementations that share a common interface are said to be “plug compatible”.

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

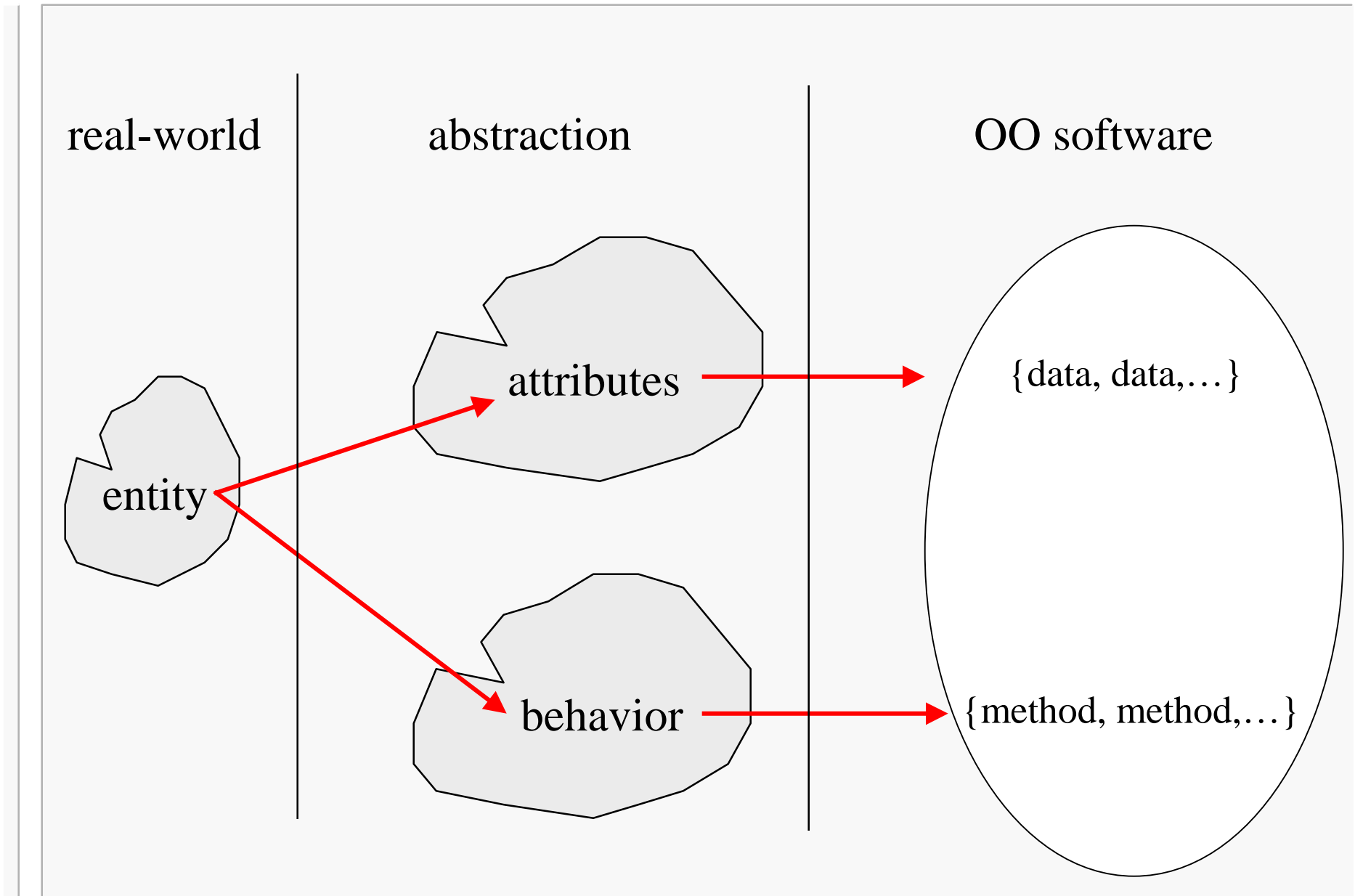
Also allows a single implementation to support multiple interfaces.

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.



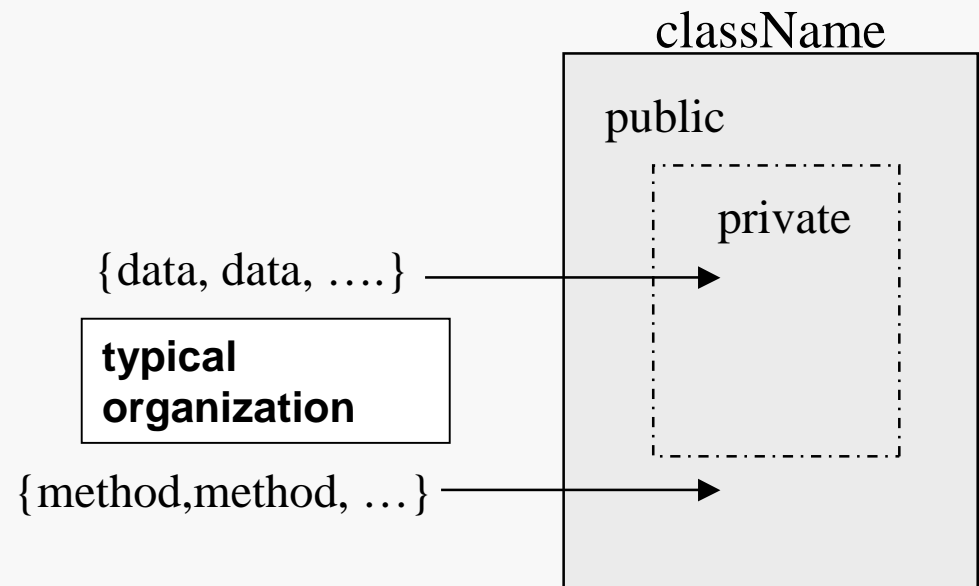
class: a named software representation for an abstraction that separates the implementation of the representation from the interface of the representation

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.



General Structure of an Object

object: a distinct instance of a given class that encapsulates its implementation details and is structurally identical to all other instances of that class

An object “encapsulates” its data and the operations that may be performed on that data.

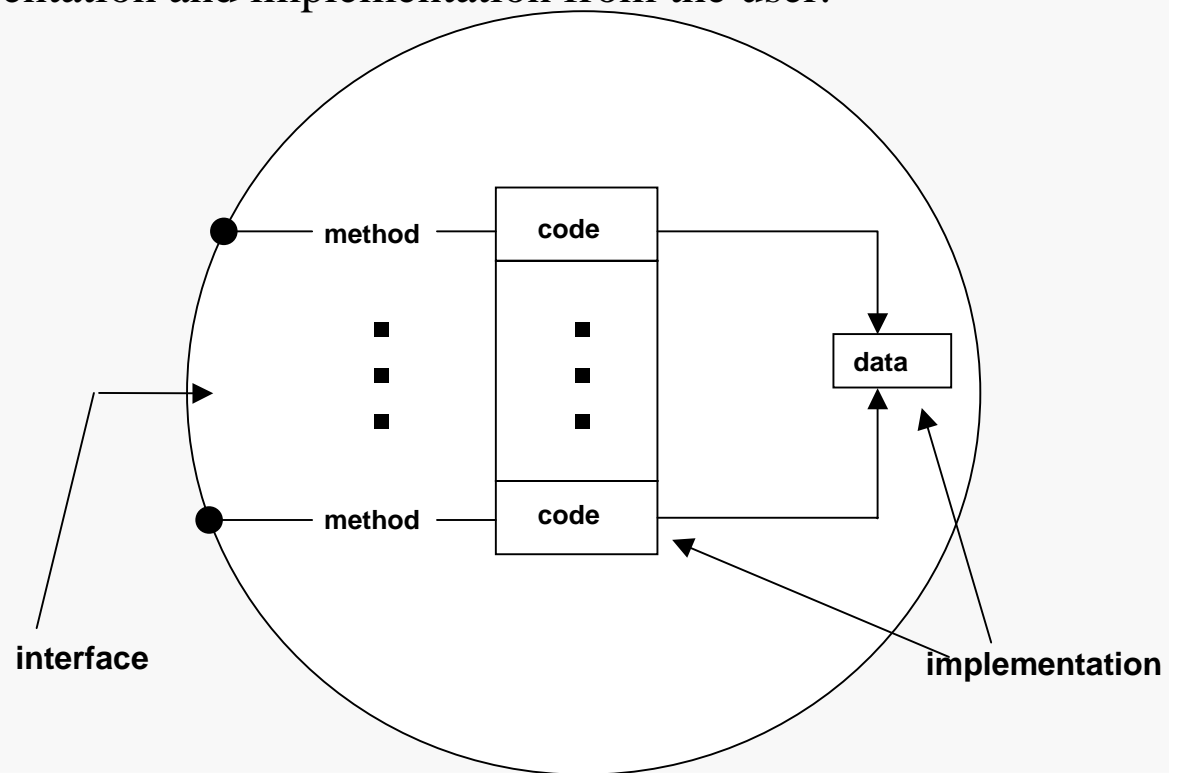
An object’s private data may **ONLY** be accessed via the member functions defined within the object’s class.

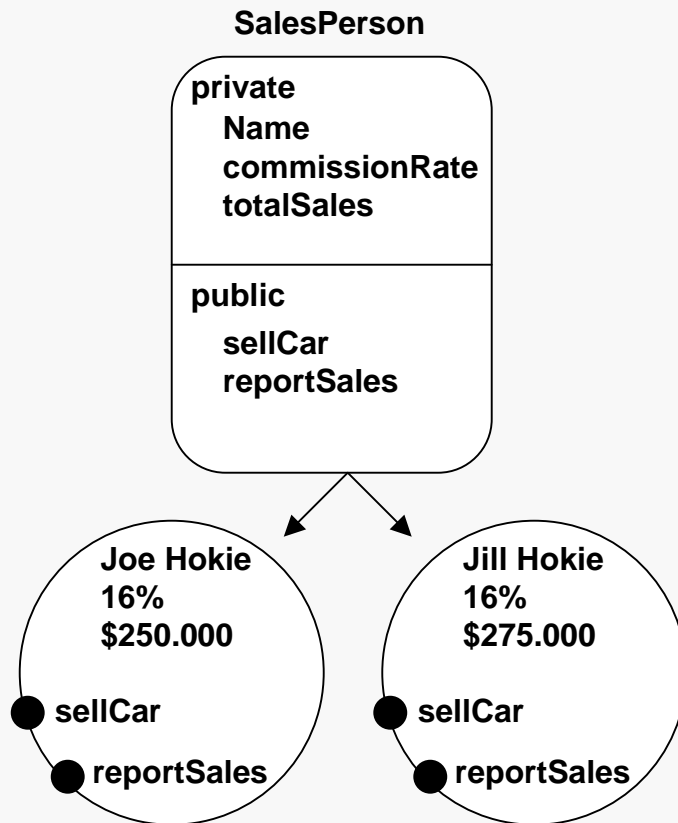
An object hides details of representation and implementation from the user.

C++ note:

Privacy restrictions are enforced at the class level, NOT the object level.

That is, if A and B are of the same type, and A knows B’s name, then A can access the private members of B directly.





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

Class = Factory Objects = Products

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

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.

