

OO Analysis and Design Guidelines

Domains To Model

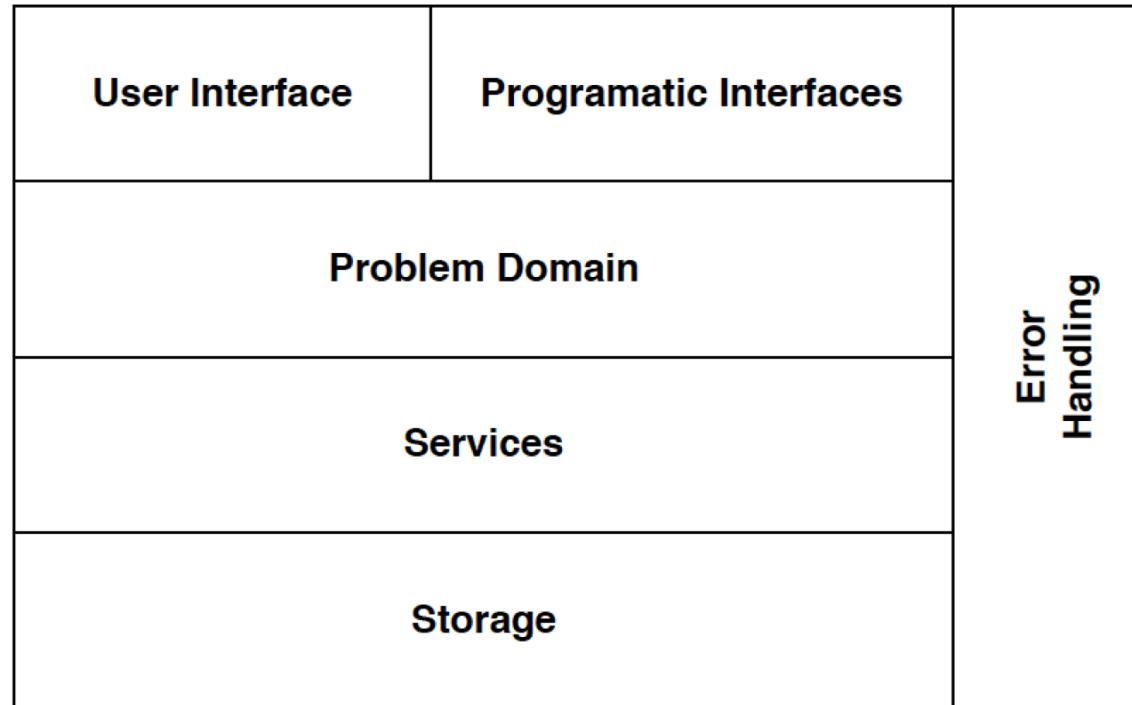
Every real-world system should be designed in several explicit domains.

- Business problem.
- Persistent storage of data.
- User interface.
- System context / network environment / security.
- Functional distribution.
- Solution *infrastructure*.

Be careful not to mix responsibilities across domains.

- Separate concerns.

Layered Architecture Model



Model Physical & Conceptual Entities

OO Programming originated as a better way to think about writing *simulators* - programs intended to behave exactly like some real physical process.

Physical:

- Domain entities: automobiles, airplanes, ...
- System entities: printers, modems, sensors, actuators, ...

Conceptual:

- Priority, Access privilege, Data format converter, Transaction manager, Memory manager, Data Structures
 - Graph / Tree / Queue / Stack / HashMap

Model Categories of Classes

- Animal
 - Dog
 - » German Shepherd
 - » Poodle
 - » Mutt
 - Cat
- Magnetic Media
 - Disk
 - Tape

Natural categories often make good inheritance hierarchies.

Model Groups of Objects

- Game Layout
 - Rows
 - » Sticks
- Corporation
 - Divisions
 - » Departments
 - ◆ Employees

Groups of objects are modeled by *composition* and *aggregation*.

Collections are special kinds of groups.

Model External Things You Must Call

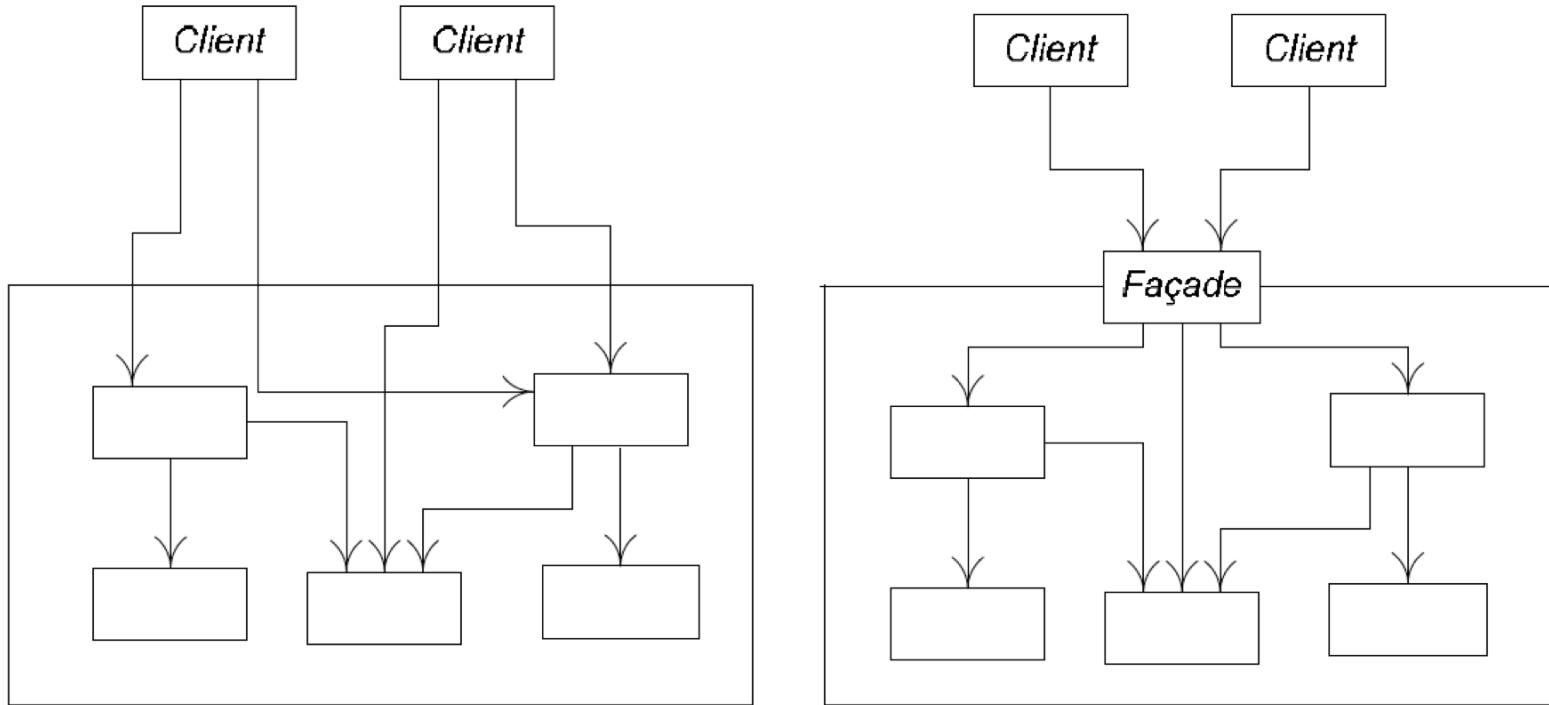
Very important in the real world.

- Examples:
 - legacy code and data sources.
 - The operating system, if you call it.
 - vendor packages.
 - system interfaces.
- Identify the services provided.
- Create objects that are responsible for providing the services.
- Consider a *proxy* object for remote services to encapsulate the distributed communications.

Design Pattern: *Façade*

- **Intent:** Provide a single interface to an architectural layer or component. The Façade class **controls** the other classes that make up the sub-system.
- Promotes ***low coupling***; the client knows only about the Façade.
- May compromise ***cohesion***; the Façade class itself can get large.
- Does little more than ***delegate*** to other classes inside the package.
- Often used in distributed applications to increase performance by reducing the number of calls across the network.
- Examples:
 - Modern interface to legacy system
 - Customer Service Representative

Façade Example



- The Façade defines an interface that makes the subsystem easier to use. Clients remain ignorant of the details of the subsystem's components.

Checking Your Classes

- Every class should have a clear name that sounds like a thing, not a function. ClassNames in Java should always begin with a capital letter and use MixedCaseLikeThis.
- Every class should have a clear purpose that applies to all of its subclasses.
- Classes that are similar but not identical might suggest inheritance, or possibly delegation to a third, shared class.
- Classes should only overlap if they have an inheritance relationship, and then one class should completely overlap the other.
 - Otherwise, classes should *separate their concerns*.
- Classes may collaborate to fulfill their responsibilities.
- Classes should have semantically related attributes and methods.

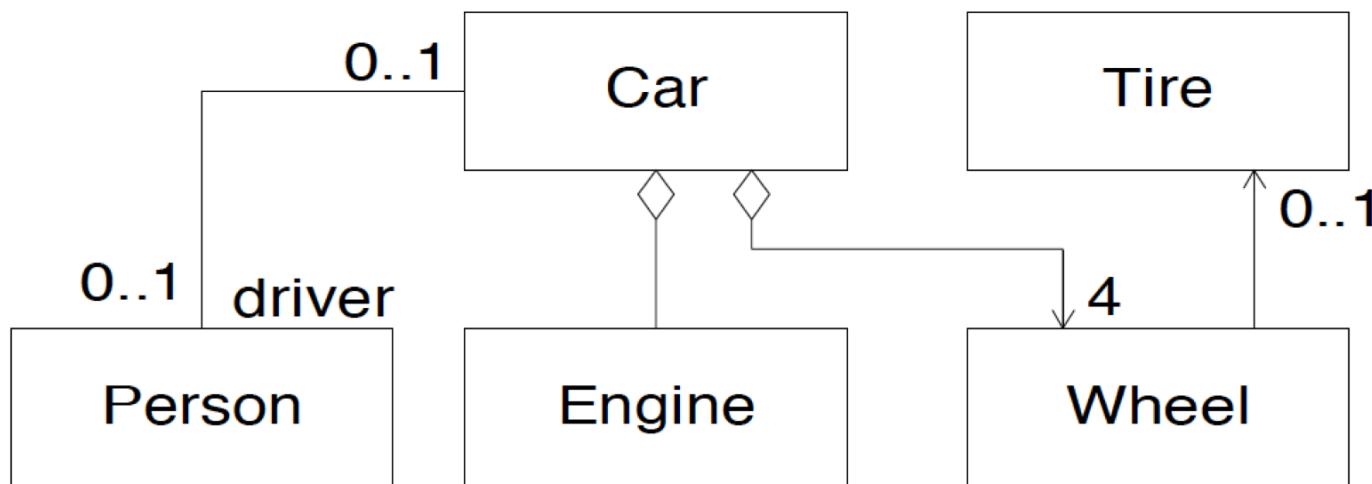
Relationship Types

- Composition
 - B *is part of* A
 - A *contains* B
 - A *has a collection of* Bs
- Subclass / Superclass
 - A *is a kind of* B
 - A *is a specialization of* B
 - A *behaves like* B
- Collaborative
 - A *delegates to* B
 - A *needs help from* B
 - A and B are *peers*.

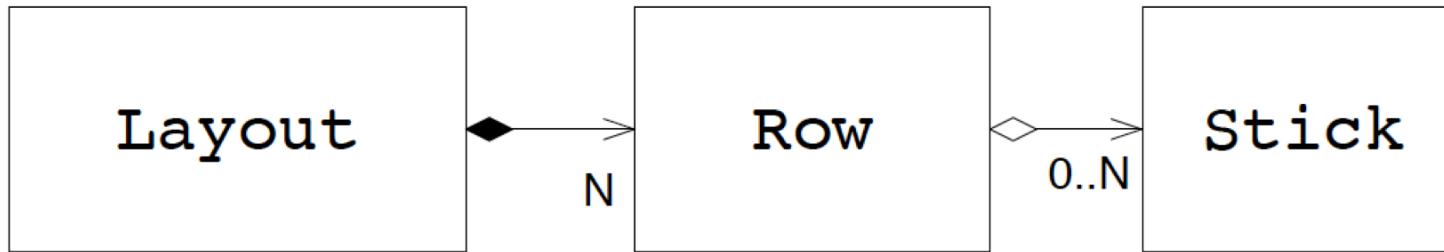
Composition / Aggregation

The diamond symbol can represent more than one concept:

- Part / whole relationships (most common)
- Has - a
- Has - a - collection - of
- Is - composed - of



Composition & Aggregation



Composition:

- UML blackens the *composition* diamond.
- The hollow diamond is used for *aggregation*.
- Composition is a stronger association than aggregation. The difference is that with composition, the part never has more than one whole, and the part and the whole always have a shared lifetime.

Composition, Aggregation, & Associations

Composition:

- A book is composed of its pages and cover.

Aggregation:

- A bookshelf holds a collection of books that changes over time.

Association:

- A book has an associated author.

Dependency:

- A person reads a book, then gives it to a friend.

Association Semantics

For Composition / Aggregation:

- Can the containee be contained within more than one container?
- Are the lifetimes of the two objects exactly the same?
- Does one object own/control the other's memory?
- Can the association be labeled *part of* or *composed of*?
- Or would it be better labeled *collection of*?

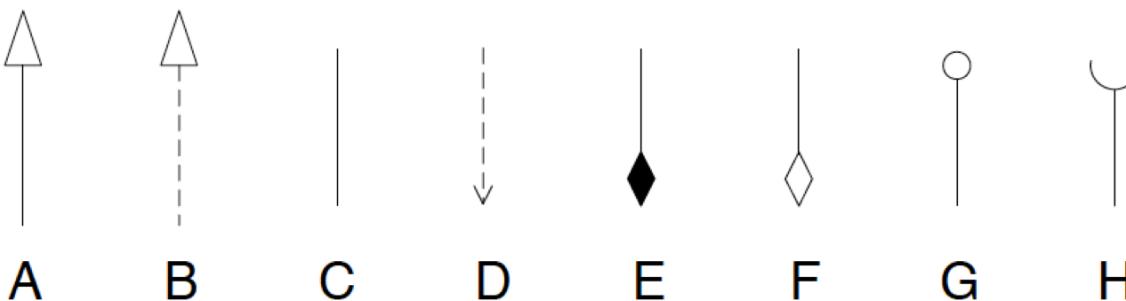
For Associations / Dependencies:

- Is the association transient, permanent, or somewhere in between?

Sometimes these distinctions are not black and white.

Consider the memory management implications (especially in C++).

UML Association Review



- A) Implementation Inheritance (Generalization)
- B) Interface Inheritance (Realization)
- C) Bidirectional Association
- D) Unidirectional Dependency
- E) Composition
- F) Aggregation
- G) Provided Interface (Lollipop)
- H) Required Interface (Socket)

Assigning Responsibilities

- Think about how the program will actually work.
- State responsibilities as generally as possible.
- Keep information about one thing in one place.
- Responsibilities can be shared or delegated.
- A class with no responsibilities is likely superfluous.
- Refactor your design whenever it gets too complicated.
- Break up big, complex classes.
- Centralized “intelligence” is inflexible.
- Avoid complexity in the graph of interacting objects.
- Poor choices lead to fragile systems.

Interfaces

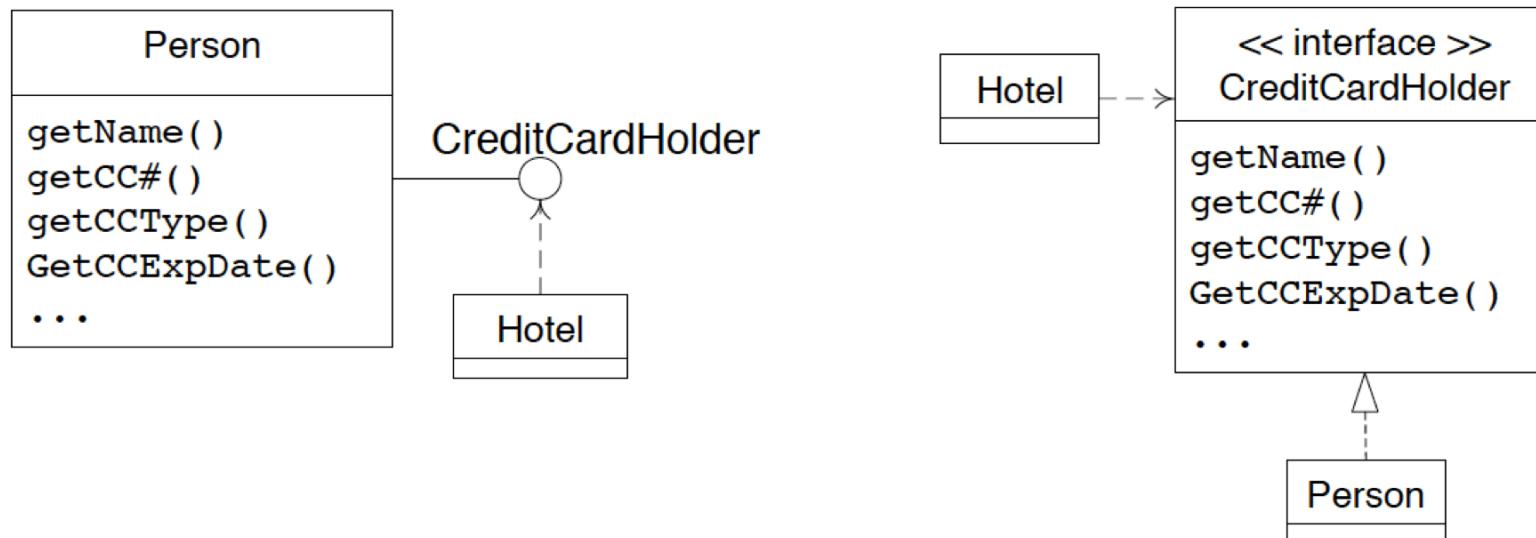
The client object sends a message to an object with a known interface; any class that implements the given interface will do.

Example: Class **Person** can implement a **CreditCardInfo** interface, used by an airline reservation program.

- The program doesn't know or care about **Person**, only about objects that implement the **CreditCardInfo** interface.
- There might also be a **Corporation** class that implements the **CreditCardInfo** interface.
- The **Person** class can change dramatically without the reservation program having to be changed at all.

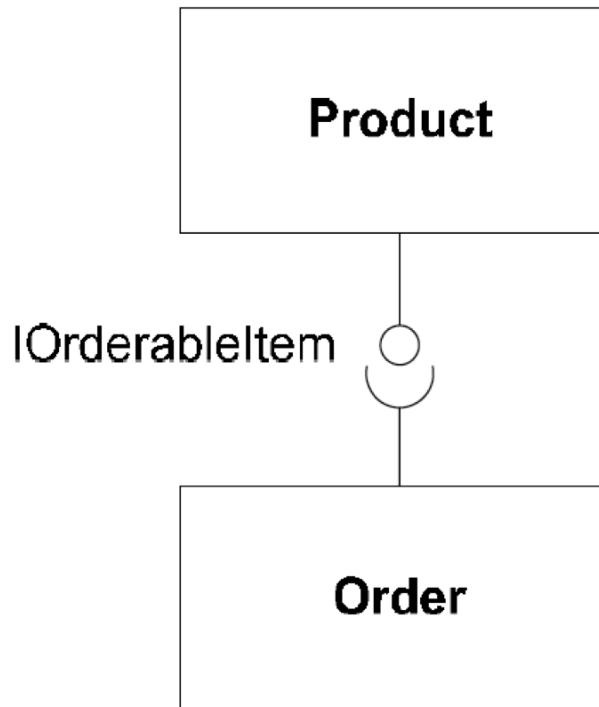
Interfaces

- Java interfaces may define *no implementation*.
- C++ interfaces are built with *purely abstract* classes.
- The use of this so-called *lollipop notation* is optional.



Interfaces and Sockets

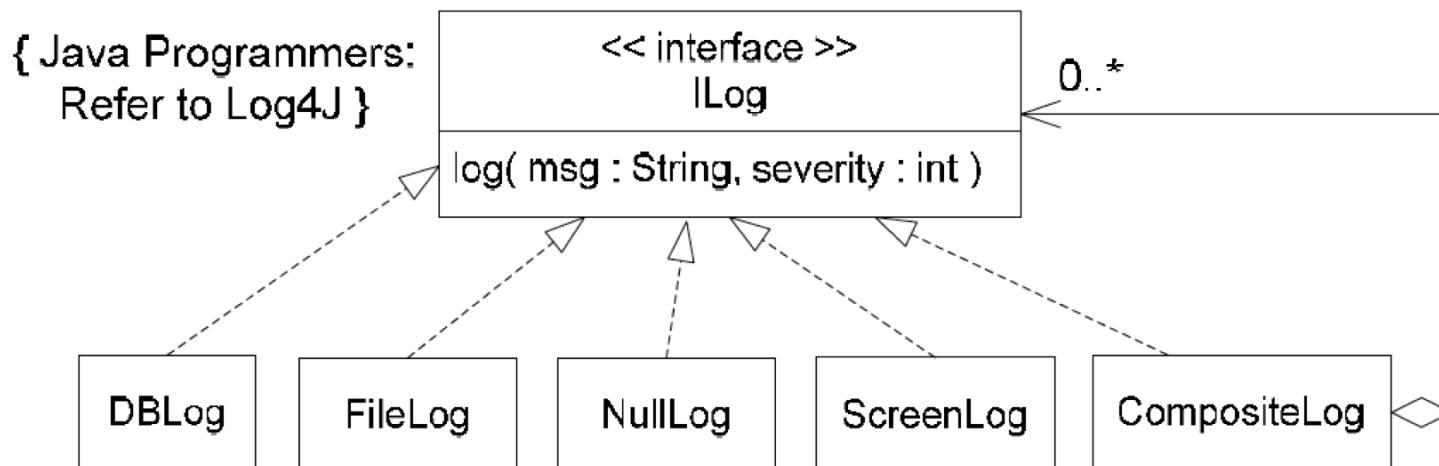
- Product is-a-kind-of IOrderableItem (it implements the interface).
- Order requires an IOrderableItem (this is called a *socket*).



Interface Example

Interfaces may be represented...

- Using the “lollipop” notation, as in the previous slide.
- As a class adorned with the <<interface>> stereotype.
- By naming convention, IWhatever.
- Note the dotted line on the inheritance relationship.



Interface Example

- You decide that your Video Store system could be used to manage other businesses that rent things (e.g., ski shops & libraries).
- You make a new abstract class called RentableObject with rent() and return() methods.
- Make Video extend RentableObject.
- This sounds easy, but... core system classes like Video may already be in a different inheritance hierarchy.
- In C++ you can use multiple implementation inheritance, but in Java you can't.
- Instead, create an interface: IRentableObject.