Object Relationships

Objects can work together in three different types of relationships:

- **Uses**: An object can use another to do some work (association).
- **Composition**: A complex object may be composed of several parts (aggregation, containment).
- **Simple**: Two objects may depend on each other but don’t interact directly (weak association).

The underlying principle behind these categories is found in the idea or **coupling**, which measures the mutual dependence of two objects on each other.

In general, you want to create classes that are as loosely coupled as possible while still being able to efficiently carry out their responsibilities.

This goal is laudable; however, most of the time the nature of the problem itself will dictate the type of association you must use.

**Cardinality**

When you design associations between objects, regardless of the type of associations, you ask yourself these questions?

- How many objects will participate in the association?
- Is the association mandatory or optional?

**Ex**: SalesInvoice
Objects that Use Objects

Whenever one object sends a message to another object (even if object doesn’t send messages back), a *uses* relationship is established them.

Ex: A SalesInvoice *uses* a Customer object

Where do you get a Customer? Six different techniques for implementing a *uses* relationship:

I. **Customers as Arguments**

Since the SalesInvoice association is a mandatory association (you can’t have a SalesInvoice without a Customer), the first option would be to pass the Customer as an argument to the constructor of the SalesInvoice. This solution lacks something in practicality. Users of the class SalesInvoice have to get hold of a fully constructed Customer before they can start.

II. **Creating Your Own Customers**

A second solution is to construct the Customer object in the SalesInvoice class constructor.

This solution may seem attractive if you’re in a business where you never get any repeat Customers but, why hold on to the Customer abstraction at all? The advantage of having Customer objects is that they exist on their own.
III. Third Party Customers

The next solution satisfies the need for Customers that can exist independently. It assumes that a repository of Customers (*CustomerDB*) keeps track of all Customer objects.

In a real application, the static method *CustomerDB.getCustomer()* would loop up a Customer in your database and return a copy of the Customer for you.

This has the advantage of working very much like the problem you’re modeling.

Additional advantage of “spinning-off” parts of the Customer abstraction that are necessary for the Customer class to work, but aren’t purely part of the Customer abstraction itself.

One of the most common methods of layering your class is to split off the user-interface code and the data-storage code into separate tightly coupled classes that work together.

This solution suffers form the same failing as the very first; it requires you to supply a valid customer number to fill out the SalesInvoice.

IV. Combining Constructors

One solution is to give the SalesInvoice class two constructors and use a no-argument constructor when you want to create a new SalesInvoice object.

Each method you’ve looked so far relies on the SalesInvoice class to store a referential attribute, *soldTo*, in its private interface. Can you create a *uses* relationship without storing such attributes? Yes, and in many cases you should.
V. Passing an Argument

Sometimes an object may have a very short and transitory relationship with another object. You may need to use an object only during the execution of a single method.

In that case, you might choose to have the method take as an argument an instance of the object it uses. This is appropriate when:

- The obj you are going to use will be used in a single method, and the state of that obj doesn’t need to be retained between calls.

- The obj you have to use carries with it some immediate state info that must be renewed each time the method is called.

- The obj you’re using is easily constructed outside your class. If, however, it’s easier for you (using class) to construct the obj from scratch (because of some special knowledge), use the objects on-the-fly technique.

- If an identical obj is constructed over and over, or it it’s very time-consuming to construct an obj and you need to use it more than once, you should use a referential attribute to hold a reference to the obj you’ll use, rather than pass it as an argument or create it on-the-fly.

VI. Creating Objects On-The-Fly

A second way to get an object that will be used in only a single method is to create it inside the method itself.

This has the advantage that it makes the calling method easier, and that the code where the object is created inside the method is inherently more maintainable.

This technique is called for when the using object has special knowledge that required to create the object that it will use.
Rule for Using Objects

You should use an attribute when:

- The obj needs to be directed from several different methods, or the obj stores persistent state info between method calls.
- The obj you are going to use is used repeatedly.
- The obj is very expensive or time consuming to construct, and you will use the obj more than once.

Pass the object as an argument when:

- The object you want to use will be used only in a single method.
- It’s easier to construct the obj outside your class. This is the case when the obj bring in some information supplied by the caller.

Construct the object on-the-fly when:

- The object will be used in only that method.
- The invoking object has info needed to construct the object that will be used, info that would be more difficult or impossible for an outside caller to supply.
Optional Associations

An optional Recipient association means that the \textit{shipTo} field contained in the \texttt{SalesInvoice} class can refer to either a valid \texttt{Customer} object or a \texttt{null} value.

What does this mean for the way you implement your class?

- The constructors for the \texttt{SalesInvoice} class don’t need to pass in a valid value for \textit{shipTo}.
- Any code that could possibly manipulate the \textit{shipTo} value must first check to see whether it’s \texttt{null}.

This is the major difficulty you’ll encounter when you want to implement optional relationships: they make the code more complex.

Weak Associations

So far no messages seem to be flowing between the \texttt{SalesPerson} and the \texttt{SalesInvoice} object. Can you just eliminate the \texttt{SalesPerson} attribute from the \texttt{SalesInvoice} class altogether?

\textbf{No.} At the end of each month or week, you’ll have to add up all the \texttt{SalesInvoices} and pay each \texttt{SalesPerson} the appropriate commission.

Should you attempt to send some messages between the \texttt{SalesPerson} object and the \texttt{SalesInvoice} object? \textbf{No.} It makes no sense to initiate meaningless messages.

A weak association exists whenever you have an association where no messages are exchanged, but the association is necessary to the success of the overall abstraction you are trying to model.
Composition

Sometimes, the best way to describe the relationship between objects is to say that one contains the other. Using composition is one of the key techniques you can use in your battle against complexity.

What is the difference between this part-of relationship and a simple uses relationship?

The distinction between the three forms of association is a semantic distinction, not an implementation distinction.

In Java, it’s likely that all three forms – weak association, uses relationships, and part-of relationship – will be represented in exactly the same way when the code is written.

Composition vs. Association

Composition is a strong form of the uses relationship. For composition to exist, either parts of the whole must send messages to each other.

Unlike a regular uses relationship, however, a composition relationship implies that one class (the whole) is made up of other objects (the parts). Composition is thus a whole-part relationship, where:

- **One object is physically built from other objects.**

  Ex: a custom UI component built from more basic components.

- **An object represents a logical whole, composed of smaller parts.**

  Ex: A company may be composed of finance, production, and marketing departments, for instance.
- **Each part belongs to only a single whole.**

  Ex: in the SalesInvoice application, a single Customer may have placed several orders, and thus be referenced on several SalesInvoices. Thus, the SalesInvoice doesn’t *contain* a Customer object.

- **Each part lives and dies with the whole.**

  Ex: in the SalesInvoice application, you certainly wouldn’t want your Customers disappearing every time you deleted a SalesInvoice object. Therefore, it is a uses relationship. You would, however, want the LineItems to disappear with the rest of the SalesInvoice when you delete it. Thus, it is a composition.

**Example:** The Labeled TextField

Java has TextField and Label objects, but it doesn’t have a built-in class that combines both.

The LTextField class will contain two private objects: a TextField called *theText*, and a FixedWidthLabel object called *theLabel*.

The most important characteristic of the LTextField class is that its parts are encapsulated as part of its state. Instead, users are presented a simple interface consisting of a *constructor*, a *set* and a *get* method.

By hiding non-essential details inside a class, and then replacing those details with a simpler interface, composition allows you to concentrate on the task at hand without getting bogged down in minutia.
Rule of Thumb for Composition

Trying out a design, noting its deficiencies, and then building a better model is an essential component of OO modeling. However, you need some way of telling whether you are making progress.

To build robust composite objects, you first have to decide which parts go with what whole. This is the main problem in composition: deciding what to put together and what to separate.

The following rules will help you tell whether you’re going in the right direction:

I. Localize Message Traffic

Objects with high-frequency communication between them generally belong in the same subsystem. Those with less communication belong in separate subsystems or modules.

Similar rules by other authors propose the following:

- If a class contains objects of another class, the container should be sending messages to the contained object.

- Most of the methods defined in the class should be using most of the data members most of the time.

II. Avoid Mixing Domains

In software development, a subsystem’s domain is the area of functionality for which the subsystem is responsible.

Layering – breaking apart an application based on common functionality – is a time tested method of grouping similar code together.
One natural form of layering, which you should definitely consider whenever you build software, is to separate the user-interface, data storage, and business logic portions of your programs into different layers.

The separation of domains is closely related to the principle of **cohesion**.

### III. Isolate the Changeable

Group objects by their **stability**, their tendency to change.

If you can, identify those portions of your system likely to change, and put those parts together. Then, combine the relatively unchangeable parts into separate subsystems.

### IV. Create Simple Interfaces

Whereas composite objects have high-frequency internal communications, communication between the “outside” world and the object should have lower frequency.

One way to facilitate this is to create simple interfaces. One consequence of this is that users can’t do all the things that they might do with the component parts.

You should, however, resist this impulse. The strength of composition lies in its capability to bring structure to complexity.

For those times when more or all needs to be visible, inheritance rather than composition is the better choice.
V. Generalize When Possible, Specialize When Necessary

Another problem you’ll encounter is deciding how general or specific each part should be.

By making very specific parts – specialized to do only a single task – you reduce the effort that a user needs to expend in using your part.

On the other hand, if you create only specialized parts, you’ll soon be drowning in classes.

Stable complex systems tend to be composed of a few simple parts, arranged in different ways. If you want your system to be maximally stable, create generalized parts whenever you can.

VI. Prefer a Tree to a Forest

Should your composite object be shallow or deep?

In a shallow containment hierarchy (forest), most composite object have many fields, and the fields are composed of relatively basic types.

In a deep containment hierarchy (tree), your system is composed of more vertical layers.

When you create deep hierarchies, you limit the amount of information you need to absorb at any one level, increasing your processing capability.
Class Relationships - Inheritance

There’s not just one, but three forms of inheritance (isA) relationships:

**An extension relationship:** A subclass may extend an existing class, so that all the data members and all the methods of the superclass are left intact, and only new methods or fields are added.

This extension or strict-subtype relationship involves the inheritance of the interface as well as its implementation.

**A specification relationship:** A superclass may specify a set of responsibilities that a subclass must fulfill, but not provide any actual implementation.

In this case, the only thing inherited is the interface from the superclass.

**A combination of extension and specification:** The subclass inherits the interface and a default implementation of, at least some of the methods (most common).

The subclass may override an inherited method to provide specialized behavior, or it may be required to provide an implementation for a particular method, which could not be provided by the superclass.

We’ll call this form **polymorphic inheritance**, because its principal value lies in its ability to provide specialized behavior in response to the same message.
Discovering Inheritance Relationships

After deciding on the classes, object, and their relationships, you are ready to begin looking at the class relationships.

You may discover that many objects share data or behavior, leading you to start at the bottom of your system and work up.

Or you may start with your more general classes and find that you need more specialized instances in certain cases, finding yourself working form the more abstract to the more concrete.

In most systems, you’ll need to do both: Your inheritance hierarchy will evolve from iterative rounds of top-down specialization and bottom-up generalization.

Specialization

The subclass is a special case (kind-of) of the parent class; in other words it is a subtype.

The new class is a specialized form of the superclass but satisfies the specifications of the base class in all relevant aspects.

    Ex:    Window, TextEditWindow
           VendingMachine, CoffeeMachine

Inheritance for Subtyping: when its possible to use descendants in all places where the base can be used.

Focusing on (object's protocol) subtyping issue will lead to proper inheritance hierarchies that are maintainable and robust. More natural from a logical perspective.
**Liskov Substitution Principle:** If for each object \( O_1 \) of type \( S \) there is an object \( O_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( O_1 \) is substituted for \( O_2 \), then \( S \) is a subtype of \( T \) (Describes total polymorphism).

- Inheritance for subtyping
- Behaviorally compatible
- Same Interface
- Inheritance performs extension; not overriding, no restriction

If every subclass object is, indeed, a SuperClass object, the subclass must uphold the implied contract that the superclass creates.

**Generalization**

Sometimes when working on an hierarchy, you’ll notice that several classes have data and methods in common.

In a procedural language, this might be an opportunity to create a new function or module.

In the object oriented world, this is a clue to look for generalization. This is the “flip-side” of specification.

**Ex:** In a catering company application, you may had created classes for CoffeMachines, SodaMachines, and CandyMachines. Later you may notice that you could generalize and factor out the commonality.
**Specification**

Sometimes, you know exactly what a subclass should do, but you simply have no idea how it should do it.

In this case, you want to specify what the subclass should do, and then let the compiler make sure that each class complies.

The base class defines behavior that is implemented in the subclass but not in the base class.

Similar to specialization, except that the subclasses are not refinements of an existing type but rather realizations of and incomplete abstract specification.

Used to guarantee a common interface.

  Ex: GraphicalObject, Ball, Wall.

**Specialization and Specification**

When you have a class that needs to specify some behavior that a subclass must perform, and yet has some behavior of its own to pass on to its subclasses. Ex: VendingMachine, dispenseItem()

**Limitation or Contraction**

When the subclass restricts the use of some of the behavior inherited. Occurs most frequently when a programmer is building on existing classes.

  Ex: DoubleEndedQueue is limited to make a Stack by modifying undesired methods.
It build subclasses that are not subtypes and should be avoided whenever possible (confusing conceptual model).

Can be avoided by using composition instead.

**Variance**: The subclass and base class are variants of each other, and the class-subclass relationship is arbitrary. Employed when two classes have similar implementations but do not seem to posses any hierarchical relationships between the abstract concepts represented by the classes.

Ex: The code required to control a mouse may be nearly identical to the code required to control a graphic table. Conceptually, however, there is no reason why Mouse should be a subclass of Tablet (or the other way).

Usually a better alternative is to factor out the common code into an abstract class (if possible), like PointingDevice.

**Combination**: The subclass inherits form more than one base class (multiple inheritance).

**The Cost of Inheritance**

- Execution Speed: Difference is often small. Increase in software development. Better to monitor and improve working system.

- Program Size: Use of program library. However, containing development costs and producing error free code rapidly is now more important than limiting the size of programs.

- Message-Passing Overhead: Increased cost is often marginal (10%).

- Program Complexity: Understanding the control flow of a program may require several multiple scans (yo-yo problem).