**Introduction**

In object-oriented software design (OOD), classes are templates for defining the characteristics and operations of an object. Often, classes and objects are used interchangeably, one synonymous with the other. In actuality, a class is a specification that an object implements.

Identifying classes can be challenging. Poorly chosen classes can complicate the application’s logical structure, reduce reusability, and hinder maintenance. This article provides a brief overview of object-oriented classes and offers tips and suggestions to identify cohesive classes.

**Note**: The following class diagrams were modeled using [Enterprise Architect](http://www.sparxsystems.com.au/ea.htm). Many other modeling tools exist. Use the one that is best suited for your purpose and project.

**Classes**

Object-oriented classes support the object-oriented principles of abstraction, encapsulation, polymorphism and reusability. They do so by providing a template, or blueprint, that defines the variables and the methods common to all objects that are based on it. Classes specify knowledge (attributes) - *they know things* - and behavior (methods) - *they do things*.

**Classes are specifications for objects.**

Derived from the [Use Cases](http://www.codeproject.com/gen/design/pusecase.asp), classes provide an abstraction of the requirements and provide the internal view of the application.

**Attributes**

Attributes define the characteristics of the class that, collectively, capture all the information about the class. Attributes should be protected by their enclosing class. Unless changed by the class’ behavior, attributes maintain their values.

The type of data that an attribute can contain is determined by its data type. There are two basic data types: Primitive and Derived.

1. Primitive data types are fundamental types. Examples are integer, string, float.
2. Derived data types are defined in terms of the Primitive data types, that is, they form new data types by extending the primitive data types. A Student class, for example, is a derived data type formed by a collection of primitive data types.

When defined in the context of a problem domain, derived data types are called Domain Specific Data types. These are the types that define and constrain attributes to be consistent with the semantics of the data. For example, Address studentAddress versus string studentAddress.

**Where practical, design models should use domain specific data types in lieu of primitive data types.**

When looking for attributes in the design model, look for these types:

1. Descriptive attributes provide intrinsic information about the class. For example, information about enrollment status, test scores, and education goals would be intrinsic to a Student class but not an Automobile class. Ask, “What characteristics distinguish this class from others?”
2. Naming attributes are used to uniquely identify an object and typically don’t change during the lifetime of the object. Consider two Student objects that contain the naming attribute, studentName. The values of “Jack” and “Jill” would identify two different Student objects. Ask, “What uniquely identifies this object from another object of the same class?”
3. Referential attributes contain links to objects. A student’s test scores might be stored in an Assessment class, for example. The value for the Assessment attribute (in the Student class) would link the Student class to the Assessment class. Ask, “How is this object associated with other objects?”

**Responsibilities**

A class has behaviors and responsibilities. A responsibility is something that a class knows or does…it’s an obligation a class has to know certain information and/or to perform a task. For example, a student knows about her address and SSN. A student does enroll in a class.

Technically, responsibilities are not the same as the class’ operations (behavior), but, operations fulfill the responsibilities. It may take more than one operation (sometimes, more than one object) to fulfill the responsibility.

**Responsibility is the obligation of a class or object to perform a task or know information.**

**Identifying classes**

Identifying object-oriented classes is both a skill and an art. It’s a process that one gets better at over time. For example, it’s not unusual for inexperienced designers to identify too many classes. Modeling too many classes results in poor performance, unnecessary complexity and increased maintenance. On the other hand, too few classes tend to increase couplings, and make classes larger and unwieldy. In general, strive for class cohesiveness where behavior is shared between multiple, related classes rather than one very large class.

**Cohesive classes reduce coupling, enable extensibility and increase maintainability.**

Moreover, classes that seem obvious wind up being poor choices and classes that are initially hidden or that rely on problem domain knowledge wind up as the best choices.

Therefore, begin class modeling by identifying candidate classes - an initial list of classes from which the actual design classes will emerge. Design classes are the classes that are modeled. Candidate classes exist for the sole purpose of deriving the design classes. Initially, there will be a lot of candidate classes – that’s good. However, through analysis, their number will be reduced as they are dropped, combined and merged.

**Candidate classes provide the initial impetus to produce cohesive classes.**

Candidate classes can be discovered in a variety of ways. Here are three:

1. [Noun and noun phrases](http://en.wikipedia.org/wiki/Noun): Identify the noun and noun phrases, verbs (actions) and adjectives (attributes) from the Use Cases, Actor-Goal List, Application Narrative and Problem Description.
2. [CRC cards](http://alistair.cockburn.us/crystal/articles/ucrcc/usingcrccards.html): an informal, group approach to object modeling.
3. [GRASP](http://cis.gsu.edu/%7Ecstucke/cis3310/3310.3.a.pdf): A formal set of principles that assign responsibilities.

Each of these methods will yield a list candidate classes. The list won’t be complete nor will every class be appropriate and there will likely be a mix of business and system oriented classes; i.e.; Student and StudentRecord, for example. That’s fine. The goal is to identify the major classes - the obvious ones. Other classes will become apparent as the design process continues.

Once the list has been created, analyze the candidate classes for associations with other classes. Look for collaborating classes. How does each relate to each other and to the business process? Sometimes, it’s helpful to ask, “Why keep this class?” In other words, assume the class is redundant or unnecessary. Keep it only if it plays a collaborating role. Often you’ll find the class’ functionality is accomplished by another class or within the context of another class.

When a class is kept, move it to the list of design classes. Eventually, a list of design classes will result that provides the foundational structure for the application.

|  |  |
| --- | --- |
| **Candidate Classes** | **Design Classes** |
| Student | Student |
| Teacher | Professor |
| Class | Course |
| Subject | Assessment |
| Grades |  |
| Tests |  |

**Design classes will emerge from the analysis of the candidate classes.**

Associations are the key to identifying cohesive classes. The following subsections identify the various associations that can exist between classes and suggestions to identify them.

**Associations**

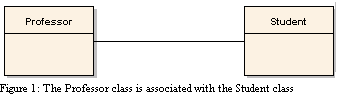
The classes in an application system don’t exist in a vacuum. Classes are associated with, or related to, other classes. These relationships occur when a class has, uses, knows about, or is acquainted with, one or more classes.

**A relationship is an association between classes.**

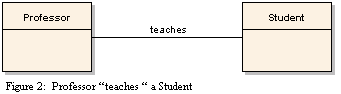
When identifying relationships, start with the class that interacts with as many other classes as possible; perhaps, the core classes of the application. It's helpful to ask, "Who cares about this class?", "Who is interested in this class?", "Why is this class necessary?" Starting with the core classes will quickly identify the other relationships.

**Start with core associations.**

An association is usually modeled using a solid line that connects two classes.

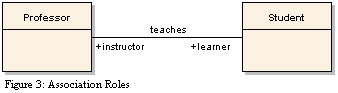


Most times, a single line doesn’t provide enough information. Form a practice of giving each association a name to clarify the relationship. Use verbs or simple verb phrases as association names.



**Association roles**

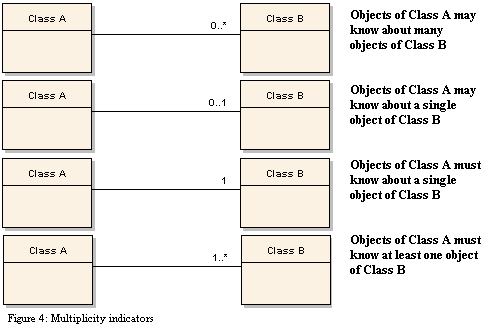
Associations also have roles. Each class in an association has a role that describes its meaning in the relationship. Roles are optional and if, used, should describe the role as a noun. For example, a Professor is an instructor to a Student who is a learner or pupil.



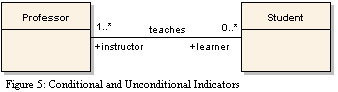
**Multiplicity Indicators**

A role can have multiplicity - an indication of how many objects participate in the relationship. Multiplicity indicators can be conditional or unconditional.

Examples of multiplicity indicators are:



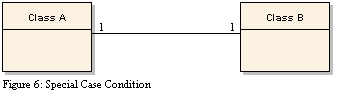
In Figure 4, the first two indicators (that start with "0...") are conditional meaning no objects need be present in the relationship. The last two indicators (that start with "1..") are unconditional meaning at least one object must be present in the relationship. For example:



In Figure 5, a Professor can exist without the existence of a Student. Therefore, the multiplicity for the Student is 0…\*. However, the same is not true for the Student. A Student must have at least one Professor. Hence, the 1…\* indicator for the Professor.

**Multiplicity indicators are also referred to as cardinalities.**

Unconditional indicators may impose a referential integrity constraint. Consider the following example:



Class A and class B depend on the existence of the other. Since the multiplicity is unconditional the following is implied:

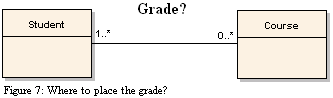
1. When class A (or class B) is removed, the corresponding class B (or class A) must also be removed.
2. When class A (or class B) is added, the corresponding class B (or class A) must also be added.

**Unconditional indicators must be checked for referential integrity constraints..**

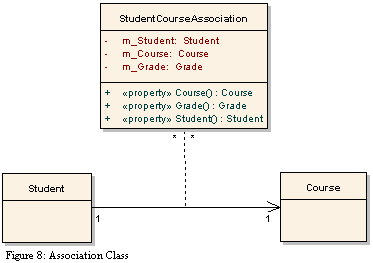
**Association class**

An association can also possess its own attributes and behavior – just like a class. Sometimes, data exist that does not strictly belong to any of the participating classes. In these cases, an Association class is created to map the data to the participating classes. The class, then, becomes the association. The association class will contain attributes that include pointers, or references, to instances of the two classes.

For example, a Student class has an association to a Course class. A student can take many courses, and a course can be taken by many students. However, who is responsible for the grade? Placing the grade in the Student class gives a student the same grade for all courses. Placing the grade in the Course class gives all students taking the same course the same grade.



The Association class resolves this issue by linking (or mapping) the grade (and other attributes) to the Student and the Course classes. Now, a Student can have many grades for many courses, but each grade in the StudentClassAssociation is associated to a single student and course.



**Association classes often occur in many-to-many associations.**

**Composition**

Sometimes a class is composed of other classes. Composition associations form a whole-part relationship. In this relationship, the life of the part depends on the whole. In other words, when the whole is destroyed or removed, the part is also. Moreover, the part cannot belong to more than one whole. This implies that the cardinality of the whole to the part is always “1”.

**In a composition association, the whole manages the lifecycle of the part.**

To identify composition associations, look for classes that can not exist without other classes. Look for "wholes" and "parts". Ask, "Is this class part of another class?" For example, a Sentence class is part of a Paragraph class that is part of a Chapter class that is part of a Book class. "Does this class depend on another class?"

**The key question to ask, "Is this class destroyed when another class is destroyed?"**

**Aggregation**

Aggregation is a weaker form of a composition. The key difference between an aggregation and a composition is, in an aggregation, the part is not destroyed when the whole is destroyed. In an aggregation relationship, the part may be independent of the whole but the whole requires the part. An automobile, for example, is composed of an engine, chassis, wheels, etc.. The wheels are required for the definition for an automobile, but, they are independent and not necessarily destroyed when the automobile is destroyed. Aggregation is often referred to as a "Has-a" relationship, as in, an Automobile Has-a(n) engine.

To identify aggregation associations, ask, "Is this class part of another class **and** is it independent of the other class?"

**To avoid confusion, when the decision to use the composition/aggregation association is unclear, it's best to model the relationship as a simple association.**

**Generalization/Specialization**

The generalization/specialization association exists when one class is a specialized version of another class. In this relationship, a common, or base, class forms the foundation for more specialized, or derived, classes. This association is commonly referred to as inheritance because the derived classes inherit the functionality of their base classes to provide specialized behavior.

**Inheritance provides the mechanism for new classes to be formed from the attributes and behavior of existing classes.**

As base classes are specialized into derived classes, hierarchies are formed that represent *is-a* or *kind-of* relationships. For example, an Algebra course *is-a* specialization of a Math course; a student is a *kind-of* Person.

To identify inheritance relationships, ask, "Is this class a specialization of a more general class?"

Beware of the tendency to see all associations in terms of inheritance. This leads to complex and deep class hierarchies that are difficult to develop. As a rule of thumb, limit hierarchies to six or fewer levels.

**Inheritance is best suited for relatively shallow class hierarchies.**

Use Inheritance when:

* The inheritance hierarchy represents an *is-a* relationship and not a *has-a* (composition/aggregate) relationship.
* The same behavior is applied to different data types.
* The class hierarchy is reasonably shallow, and unlikely to become much deeper over time.

**Interface inheritance**

Closely related to inheritance is the concept of interface inheritance. Like an inherited class, an interface provides a common specification for behavior, but, unlike an inherited class, an interface cannot be created. An interface simply specifies common behavior that other classes inherit but implement differently. This implies that unrelated classes can provide independent implementations for the same behavior specification (polymorphism).

For example, the .NET framework provides the IComparable interface that defines a generalized comparison method. This interface is typically used for sorting purposes. Classes that inherit the IComparable interface implement behavior specific to their needs. So, using the same IComparable interface, an Integer class can sort integers, a Byte class can sort bytes, a Student class can sort students, and so on.

**The behavior has been abstracted out from any particular class and specifically implemented by all classes that need it.**

Use interface inheritance when:

* Unrelated object types need to provide the same behavior.
* Multiple inheritance is needed (very difficult to implement using the inheritance association).
* Inheritance is prohibited. For example, C# structures cannot inherit from classes, but they can implement interfaces.

**The Strategy Design Pattern is an excellent candidate for an interface inheritance.**

**Conclusion**

Sometimes, developers don’t know where or how to begin identifying classes. Often classes are simply chosen based on what seems to make sense to the developer without regarding class associations or cohesiveness. As a result, the logical structure for an application can contain classes that are unnecessary and complex making the application hard to extend and maintain. This article provided tips and guidance to identify object-oriented, cohesive classes that work together to accomplish the application’s business function.

**Additional links**

* [Allen Holub's UML Quick Reference](http://www.holub.com/goodies/uml/)
* Scott Ambler’s [How to Create Class Diagrams](http://www.agilemodeling.com/artifacts/classDiagram.htm#DesignClassDiagrams).
* Scott Ambler’s [UML Class Diagram Guidelines](http://www.agilemodeling.com/style/classDiagram.htm).
* [OMG’s UML Resource Page](http://www.uml.org/).
* Craig Larman’s, [Applying UML and Patterns](http://www.amazon.com/exec/obidos/tg/detail/-/0131489062/qid=1101839515/sr=1-1/ref=sr_1_1/002-7792023-3440049?v=glance&s=books) is a must read.

FROM <https://stackoverflow.com/questions/25869391/why-do-we-use-usecase-diagrams-in-object-oriented-analysis-and-design-even-if-us#25871076>

Object-oriented is analysis and design methodology, while use case is requirements methodology. And be aware of the core development workflow：

1. Business modeling
2. Requirements
3. Analysis
4. Design

If we use UML to do these works, we may have:

1. Business use case+ Business sequence diagram
2. System use case+ System use case specification
3. Analysis class diagram+ Analysis sequence diagram+ Analysis state machine diagram
4. Code, Database......

UML diagrams in **3.** can be replaced by DFD/ER