**Association, Aggregation, Composition, Abstraction, Generalization, Realization, Dependency**

These terms signify the relationships between classes. These are the building blocks of object oriented programming and very basic stuff. But still for some, these terms look like Latin and Greek. Just wanted to refresh these terms and explain in simpler terms.

**Association**

Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

http://javapapers.com/wp-content/uploads/2010/06/association.jpg

***Example:*** A Student and a Faculty are having an association.

**Aggregation**

Aggregation is a special case of association. A directional association between objects. When an object ‘has-a’ another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a “Has-a” relationship.

http://javapapers.com/wp-content/uploads/2010/06/aggregation.jpg

**Composition**

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

http://javapapers.com/wp-content/uploads/2010/06/composition.jpg

***Example:*** A class contains students. A student cannot exist without a class. There exists composition between class and students.

**Difference between aggregation and composition**

Composition is more restrictive. When there is a composition between two objects, the composed object cannot exist without the other object. This restriction is not there in aggregation. Though one object can contain the other object, there is no condition that the composed object must exist. The existence of the composed object is entirely optional. In both aggregation and composition, direction is must. The direction specifies, which object contains the other object.

***Example:***A Library contains students and books. Relationship between library and student is aggregation. Relationship between library and book is composition. A student can exist without a library and therefore it is aggregation. A book cannot exist without a library and therefore its a composition. For easy understanding I am picking this example. Don’t go deeper into example and justify relationships!

**Abstraction**

Abstraction is specifying the framework and hiding the implementation level information. Concreteness will be built on top of the abstraction. It gives you a blueprint to follow to while implementing the details. Abstraction reduces the complexity by hiding low level details.

***Example:*** A wire frame model of a car.

**Generalization**

Generalization uses a “is-a” relationship from a specialization to the generalization class. Common structure and behaviour are used from the specializtion to the generalized class. At a very broader level you can understand this as inheritance. Why I take the term inheritance is, you can relate this term very well. Generalization is also called a “Is-a” relationship.

http://javapapers.com/wp-content/uploads/2010/06/generalization.jpg

***Example:*** Consider there exists a class named Person. A student is a person. A faculty is a person. Therefore here the relationship between student and person, similarly faculty and person is generalization.

**Realization**

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

http://javapapers.com/wp-content/uploads/2010/06/realization.jpg

***Example:*** A particular model of a car ‘GTB Fiorano’ that implements the blueprint of a car realizes the abstraction.

**Dependency**

Change in structure or behaviour of a class affects the other related class, then there is a dependency between those two classes. It need not be the same vice-versa. When one class contains the other class it this happens.

http://javapapers.com/wp-content/uploads/2010/06/dependency.jpg

***Example:*** Relationship between shape and circle is dependency.

**Design Patterns**

At an OOPSLA workshop held in 1991, Gamma and Helm were joined by Ralph Johnson and John Vlissides. This Gang of Four (GoF), as they subsequently were known, went on to write the popular Design Patterns: Elements of Reusable Object-Oriented Software, which documents 23 design patterns in three categories.

### The modern evolution of design patterns

Design patterns have continued to evolve since the original GoF book, especially as software developers have confronted new challenges related to changing hardware and application requirements.

In 1994, a U.S.-based non-profit organization known as the [Hillside Group](http://hillside.net/) inaugurated Pattern Languages of Programs, a group of annual conferences whose aim is to develop and refine the art of software design patterns. These ongoing conferences have yielded many examples of domain-specific design patterns. For example, design patterns in a concurrency context.

In 1998 Mark Grand released Patterns in Java. This book included design patterns not found in the GoF book, including concurrency patterns. Grand also used the Unified Modeling Language (UML) to describe design patterns and their solutions. The book's examples were expressed and described in the Java language.

## Software design patterns by classification

Modern software design patterns are broadly classified into four categories based on their use: creational, structural, behavioral, and concurrency. I'll discuss each category and then list and describe some of the prominent patterns for each one.

#### Other types of design patterns

If you're thinking that there are more types of patterns, you are right. A later article in this series will discuss additional design pattern types: lnteraction, architectural, organizational, and communication/presentation patterns.

### Creational patterns

A creational pattern abstracts the process of instantiation, separating how objects are created, composed, and represented from the code that relies on them. Class creational patterns use inheritance to vary the classes that are instantiated, and object creational patterns delegate instantiation to other objects.

* [**Abstract factory**](http://www.javaworld.com/community/node/3776): This pattern provides an interface to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.
* [**Builder**](http://www.javaworld.com/javaworld/jw-04-2007/jw-04-nested-classes.html): Separates the construction of a complex object from its representation, enabling the same construction process to create various representations. Abstracting the steps of object construction allows different implementations of the steps to construct different representations of the objects.
* [**Factory method**](http://www.javaworld.com/javaqa/2001-05/02-qa-0511-factory.html): Defines an interface for creating an object, but lets subclasses decide which class to instantiate. This pattern lets a class defer instantiation to subclasses. [Dependency injection](http://www.javaworld.com/community/node/606) is a related pattern. (See [Resources](http://www.javaworld.com/javaworld/jw-11-2012/121121-introduction-to-design-patterns-part-1.html?page=4#resources).)
* **Lazy initialization**: This pattern gives us a way to delay object creation, database lookup, or another expensive process until the first time the result is needed.
* **Multiton**: Expands on the singleton concept to manage a map of named class instances as key-value pairs, and provides a global point of access to them.
* [**Object pool**](http://www.javaworld.com/jw-06-1998/jw-06-object-pool.html): Keep a set of initialized objects ready to use, rather than be allocated and destroyed on demand. The intent is to avoid expensive resource acquisition and reclamation by recycling objects that are no longer in use.
* **Prototype**: Specifies the kinds of objects to create using a prototypical instance, then create new objects by copying this prototype. The prototypical instance is cloned to generate new objects.
* **Resource acquisition is initialization**: This pattern ensures that resources are automatically and properly initialized and reclaimed by tying them to the lifespan of suitable objects. Resources are acquired during object initialization, when there is no chance of them being used before they are available, and released with the destruction of the same objects, which is guaranteed to take place even in the case of errors.
* [**Singleton**](http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html): Ensures that a class has only one instance and provides a global point of access to this instance.

### Structural patterns

A structural pattern teaches us how to compose classes and objects to form larger structures. A structural class pattern relies on inheritance to compose a resulting interface or implementation (for example, multiple inheritance mixes two or more classes into one class). A structural object pattern composes various objects to obtain new functionality; the [Composite pattern](http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html) is one example of this approach

* [**Adapter**](http://www.javaworld.com/javaworld/jw-09-2003/jw-0926-designpatterns.html): Converts a class's interface into another interface that clients expect. An adapter lets classes work together that otherwise couldn't due to incompatible interfaces. It does so by providing its interface to clients while using the original class interfaces internally. Adapter is also known as the Wrapper pattern.
* **Bridge**: Decouples an abstraction from its implementation, which lets the two vary independently. The bridge uses encapsulation, aggregation, and can also use inheritance to separate responsibilities into different classes.
* [**Composite**](http://www.javaworld.com/javaworld/jw-09-2002/jw-0913-designpatterns.html): Composes objects into tree structures that represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
* [**Decorator**](http://www.javaworld.com/javaworld/jw-04-2004/jw-0412-decorator.html): Dynamically attaches additional responsibilities to an object while maintaining the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
* [**Facade**](http://www.javaworld.com/javaworld/jw-05-2003/jw-0530-designpatterns.html): Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. Facades can wrap poorly-designed APIs in a single well-defined API.
* [**Flyweight**](http://www.javaworld.com/javaworld/jw-07-2003/jw-0725-designpatterns.html): Uses sharing to support large numbers of similar objects efficiently. A flyweight is an object that minimizes memory use by sharing as much data as possible with other similar objects. It offers a way to use objects in large numbers when a simple repeated representation would result in an unacceptable amount of allocated memory.
* [**Front controller**](http://www.javaworld.com/javaworld/jw-10-2001/jw-1019-dispatcher.html): Provides a centralized entry point for handling requests. This pattern relates to the design of Web applications.
* **Module**: Implements the concept of software modules, defined by modular programming, in a programming language that does not support or only partly supports modules.
* [**Proxy**](http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html): Provides a surrogate or placeholder object for another object. This proxy controls access to the other object.

### Behavioral patterns

Behavioral patterns focus on algorithms and the assignment of responsibilities between objects. They address object or class patterns as well as the communication patterns between them. A behavioral class pattern uses inheritance to distribute behavior among classes. In contrast, a behavioral object pattern uses object composition.

* **Blackboard**: This is a generalization of the Observer pattern that supports multiple readers and writers. The Blackboard pattern communicates information systemwide.
* **[Chain of responsibility](http://www.javaworld.com/javaworld/jw-08-2003/jw-0829-designpatterns.html)**: Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. This pattern chains together receiving objects and passes a request along the chain until an object handles the request.
* [**Command**](http://www.javaworld.com/javaworld/jw-06-2002/jw-0628-designpatterns.html): Uses an object to encapsulate all information needed to call a method at a later time. This information includes the method name, the object that owns the method, and values for the method parameters. A client instantiates the command object and provides the information required to call the method. The invoker decides when the method should be called. Finally, the receiver is an instance of the class that contains the method's code.
* **Interpreter**: Given a language, defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
* [**Iterator**](http://www.javaworld.com/javatips/jw-javatip38.html): Enables developers to access the elements of an aggregate object sequentially without exposing its underlying representation.
* **Mediator**: Defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
* **Memento**: Without violating encapsulation, captures and externalizes an object's internal state so that the object can be subsequently restored to this state.
* **Null Object**: Avoids null references by providing a default object.
* [**Observer**](http://www.javaworld.com/javaworld/jw-03-2003/jw-0328-designpatterns.html): Defines a one-to-many dependency between objects where a state change in one object results in all of its dependents being notified and updated automatically. Observer is also known as the Publish-subscribe pattern.
* **Servant**: Defines functionality for a group of classes without defining that functionality in each of those classes. A servant is a class whose instance (or the class itself) provides methods that perform a desired service, while objects for which (or with whom) the servant does something are passed to servant methods as parameters.
* **Specification**: Recombines business rules by chaining the business rules together using Boolean logic.
* **State**: Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
* [**Strategy**](http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-designpatterns.html): Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
* **Template method**: Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
* [**Visitor**](http://www.javaworld.com/javatips/jw-javatip98.html): Represents an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

### Concurrency patterns

Finally, a concurrency pattern addresses some aspect of multithreaded programming. One well-known pattern in this category is Producer-consumer, in which a producer thread stores an item in a shared buffer and a consumer thread retrieves this item. The producer thread must not store another item in this buffer until the previous item has been consumed, and the consumer thread must not consume a non-existent item.

* [**Active object**](http://www.javaworld.com/jw-06-1999/jw-06-toolbox.html): Decouples method execution from method invocation for objects in which each object resides in its own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests
* **Balking**: Only executes an action on an object when the object is in a particular state. For example, an object reads a file and provides methods to access file content. When the file is not open and an attempt is made to call a method to access content, the object "balks" at the request.
* **Binding properties**: Combines multiple observers to force properties in different objects to be synchronized or coordinated in some way.
* **Double-checked locking**: Reduces the overhead of acquiring a lock by first testing the locking criterion (the "lock hint") without actually acquiring the lock. The lock is acquired only when the locking criterion check indicates that locking is required.
* [**Event-based asynchronous**](http://www.javaworld.com/javaworld/jw-03-2008/jw-03-asynchhttp.html): A concurrency pattern for the asynchronous invocation of an object's potentially long-running methods.
* **Guarded suspension**: Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.
* **Lock**: A mechanism to temporarily make some aspect of an object unmodifiable or to suppress unneeded update notifications.
* **Messaging design pattern (MDP)**: Allows the interchange of information (that is, messages) between components and applications.
* [**Monitor object**](http://www.javaworld.com/javaworld/jw-10-2007/jw-10-monitors.html): An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.
* [**Reactor**](http://www.javaworld.com/jw-06-1999/jw-06-toolbox.html): An event-handling pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.
* [**Read-write lock**](http://www.javaworld.com/jw-04-1999/jw-04-toolbox.html): A lock that allows concurrent read access to an object, but requires exclusive access for write operations.
* [**Scheduler**](http://www.javaworld.com/javaworld/jw-07-2002/jw-0703-java101.html): A concurrency pattern used to explicitly control when threads may execute single-threaded code; for example, multiple threads wanting to write data to the same file.
* [**Thread pool**](http://www.javaworld.com/jw-05-1999/jw-05-toolbox.html): A concurrency pattern in which a number of threads are created to perform various tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Thread Pool can be considered a special case of the Object Pool creational pattern.
* **Thread-specific storage**: A concurrency pattern in which static or global memory is localized to a thread. Each thread has its own copy of this memory.