<table>
<tr><th colspan="3">Creational Pattern</th></tr>
<tr><td>Factory Pattern:</td><td colspan="2">Intention:
creates objects without exposing the instantiation logic to the client.
refers to the newly created object through a common interface
E.g. Consider an example of using multiple database servers like SQL Server and Oracle. If you are developing an application using SQL Server database as backend, but in future need to change backend database to oracle, you will need to modify all your code</td></tr>
<tr><td>Abstract Factory Pattern</td><td colspan="2">Intention:
Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes.
E.g. Dependency Injection
At the root of the application we wire up all necessary object graphs. This place is called the Composition Root, and we can use a DI Container to do this wiring for us, or we can do it manually (Pure DI).
The Abstract Factory pattern is a very useful pattern when it comes to DI. In essence, use Abstract Factory when:
You need to supply one or more parameters only known at run-time before you can resolve a dependency.
The lifetime of the dependency is conceptually shorter than the lifetime of the consumer.</td></tr>
<tr><td>Singleton Pattern</td><td>Intention:
Ensure that only one instance of a class is created.
Provide a global point of access to the object.
E.g. Typically singletons are used for global configuration. The simplest example would be LogManager - there's a static LogManager.getLogManager() method, and a single global instance is used.

Using Enum: (No problem of Reflection/Serialization), but Eager Loading
public enum EnumSingleton {
INSTANCE;
public static void doSomething(){
//do something
}
}</td><td>Classic Java Singleton class:
public class Singleton {
private static Singleton instance;
private Singleton (){}

public static Singleton getInstance() {
if (instance == null) {
instance = new Singleton();
}
return instance;
}
}</td></tr>
<tr><td></td><td>ThreadSafe Singleton class:
public synchronized static Singleton getInstance() {
if(singleton == null) {
singleton = new Singleton();
}
return singleton;
}

Synchronize the critical code only:
public static Singleton getInstance() {
if(singleton == null) {
synchronized(Singleton.class) {
singleton = new Singleton();</td><td>Double checked locking:
public static Singleton getInstance() {
if(singleton == null) {
synchronized(Singleton.class) {
if(singleton == null) {
singleton = new Singleton();
}
}
}
return singleton;
}

Double checked Locking with volatile keyword:
public class Singleton {</td></tr>
</table>

	} } return singleton; }	private volatile static Singleton instance; …… }

	ThreadSafe but not lazy initialization:
	public class Singleton{
	//the variable will be created when the class is loaded
	private static final Singleton instance = new Singleton();
	private Singleton(){}
	public static Singleton getInstance(){
	return instance;
	}
	}
		Ultimate ThreadSafe and lazy initialization In Java:
		@ThreadSafe
		class Singleton {
		private static class SingletonHolder {
		public static Singleton instance = new Singleton();
		}
		
		public static Singleton getInstance() {
		return SingletonHolder.instance;
		}
		}

Builder Pattern:	● builds a complex object using simple objects and using a step by step approach E.g. JobBuilder and TriggerBuilder implementations in the Quartz scheduler package.	
Prototype Pattern:	● creating duplicate object while keeping performance in mind. ● The Prototype pattern is a creation pattern based on cloning a pre-configured object. The idea is that you pick an object that is configured for either the default or in the ballpark of some specific use case and then you clone this object and configure to your exact needs. E.g. A possible real world application might be say, when you need to create a spreadsheet containing many cells. Rather than set the style for each newly created cell to override the default stylings, you'd use a Prototype pattern to create a template cell, and clone that cell when creating new cells.	
Structural Patterns:		
Adapter Pattern	● a single class which is responsible to join functionalities of independent or incompatible interfaces. E.g. In order to connect power, we have different interfaces all over the world. Using Adapter we can connect easily likewise.	
Bridge Pattern	● decouples implementation class and abstract class by providing a bridge structure between them. ● The Bridge pattern is a composite of the Template and Strategy patterns. ● At first sight, the Bridge pattern looks a lot like the Adapter pattern in that a class is used to convert one kind of interface to another. However, the intent of the Adapter pattern is to make one or more classes' interfaces look the same as that of a particular class. The Bridge pattern is designed to separate a class's interface from its implementation so you can vary or replace the implementation without changing the client code.	
Filter Pattern	● enables developers to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations. E.g. ServletFilter	
Composite Pattern	● treat a group of objects in similar way as a single object. ● Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. E.g. In the first phase (Restore View phase) and the last phase (Render Response phase), the UI View is constructed using JSF UI components. UIComponentBase represents an abstract Component class in the Composite pattern.	
Decorator Pattern	● allows a user to add new functionality to an existing object without altering its structure. E.g. Buffered io - java.io.FileWriter and java.io.BufferedWriter both extend java.io.Writer. java.io.BufferedWriter is	

	composite and FileWriter is leaf. BufferedWriter adds additional responsibility (or feature) of buffering to FileWriter. write() method is unified interface, whereas buffering is additional feature.
Facade Pattern	● hides the complexities of the system and provides an interface to the client using which the client can access the system. E.g. Hotwire.com (Facade) , subsystems => flight , hotel, cars
Flyweight:	● used to reduce the number of objects created and to decrease memory footprint and increase performance. E.g. Cache
Proxy Pattern	● a class represents functionality of another class. E.g. Spring Transactional proxies
Behavior Patterns	
Chain of responsibility Pattern	● creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. E.g. Interceptors in Struts 2
Command Pattern	● A request is wrapped under an object as command and passed to invoker object. E.g. Hibernate Query Language. Query is wrapped so that it can be executed on any database.
Interpreter Pattern	● a way to evaluate language grammar or expression. E.g. OGNL, Regex
Iterator Pattern	● to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation. E.g. Iterator
Mediator Pattern	● provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling. E.g. ChatRoom
Memento Pattern:	● Memento pattern uses three actor classes. ● Memento contains state of an object to be restored. ● Originator creates and stores states in Memento objects. ● Caretaker object is responsible to restore object state from Memento. E.g. Queries executed on database have different states.
Observer Pattern	● is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. E.g. EventObservers
State Pattern	● we create objects which represent various states and a context object whose behavior varies as its state object changes. E.g. Traffic Light System, States: RED, Yellow, Green. Behavior: Vehicle movement
Strategy	● we create objects which represent various strategies and a context object whose behavior varies as per

Pattern	its strategy object. E.g. Sorting a list. The strategy is the comparison used to decide which of two items in the list is "First"
Template Pattern	● an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override ● the method implementation as per need but the invocation is to be in the same way as defined by an abstract class. E.g. Angular
Visitor Pattern:	● execution algorithm of element can vary as and when visitor varies. E.g. Modern Browsers. User vs Admin
MVC Pattern	This pattern is used to separate application's concerns. ● Model - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes. ● View - View represents the visualization of the data that model contains. ● Controller - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.
Business delegate Pattern	Business Delegate Pattern is used to decouple presentation tier and business tier. ● Client - Presentation tier code may be JSP, servlet or UI java code. ● Business Delegate - A single entry point class for client entities to provide access to Business Service methods. ● LookUp Service - Lookupservice object is responsible to get relative business implementation and provide business object access to business delegate object. ● Business Service - Business Service interface. Concrete classes implement this business service to provide actual business implementation logic. E.g. Microservices
DAO Pattern	Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. ● Data Access Object Interface - This interface defines the standard operations to be performed on a model object(s). ● Data Access Object concrete class - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism. ● Model Object or Value Object - This object is simple POJO containing get/set methods to store data retrieved using DAO class. E.g. ORM frameworks
Front controller Pattern	The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. ● Front Controller - Single handler for all kinds of requests coming to the application (either web based/ desktop based). ● Dispatcher - Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler. ● View - Views are the object for which the requests are made. E.g. Spring dispatcher servlet
Interceptin g filter pattern	when we want to do some pre-processing / post-processing with request or response of the application. Filters are defined and applied on the request before passing the request to actual target application ● Filter - Filter which will performs certain task prior or after execution of request by request handler. ● Filter Chain - Filter Chain carries multiple filters and help to execute them in defined order on target. ● Target - Target object is the request handler ● Filter Manager - Filter Manager manages the filters and Filter Chain.

	• Client - Client is the object who sends request to the Target object. E.g. Servlet Filters.

CQRS

- Command Query Responsibility Segregation
- At its heart is the notion that you can use a different model to update information than the model you use to read information.

Solution:

- The change that CQRS introduces is to split that conceptual model into separate models for update and display, which it refers to as Command and Query respectively following the vocabulary of CommandQuerySeparation. The rationale is that for many problems, particularly in more complicated domains, having the same conceptual model for commands and queries leads to a more complex model that does neither well.

How?

- By separate models we most commonly mean different object models, probably running in different logical processes, perhaps on separate hardware. A web example would see a user looking at a web page that's rendered using the query model. If they initiate a change that change is routed to the separate command model for processing, the resulting change is communicated to the query model to render the updated state.
- There's room for considerable variation here. The in-memory models may share the same database, in which case the database acts as the communication between the two models. However they may also use separate databases, effectively making the query-side's database into a real-time ReportingDatabase. In this case there needs to be some communication mechanism between the two models or their databases.
- The two models might not be separate object models, it could be that the same objects have different interfaces for their command side and their query side, rather like views in relational databases. But usually when I hear of CQRS, they are clearly separate models.

When to Use?

- Like any pattern, CQRS is useful in some places, but not in others. Many systems do fit a CRUD mental model, and so should be done in that style. CQRS is a significant mental leap for all concerned, so shouldn't be tackled unless the benefit is worth the jump. While I have come across successful uses of CQRS, so far the majority of cases I've run into have not been so good, with CQRS seen as a significant force for getting a software system into serious difficulties.

Problem:

- The mainstream approach people use for interacting with an information system is to treat it as a CRUD datastore.
- As our needs become more sophisticated we steadily move away from that model. We may want to look at the information in a different way to the record store, perhaps collapsing multiple records into one, or forming virtual records by combining information for different places.
- On the update side we may find validation rules that only allow certain combinations of data to be stored, or may even infer data to be stored that's different from that we provide.
- As this occurs we begin to see multiple representations of information. When users interact with the information they use various presentations of this information, each of which is a different representation. Developers typically build their own conceptual model which they use to manipulate the core elements of the model. If you're using a Domain Model, then this is usually the conceptual representation of the domain. You typically also make the persistent storage as close to the conceptual model as you can.
- This structure of multiple layers of representation can get quite complicated, but when people do this they still resolve it down to a single conceptual representation which acts as a conceptual integration point between all the presentations.

Where?

CQRS naturally fits with some other architectural patterns.

- As we move away from a single representation that we interact with via CRUD, we can easily move to a task-based UI.
- CQRS fits well with event-based programming models. It's common to see CQRS system split into separate services communicating with Event Collaboration. This allows these services to easily take advantage of Event Sourcing.
- Having separate models raises questions about how hard to keep those models consistent, which raises the likelihood of using eventual consistency.
- For many domains, much of the logic is needed when you're updating, so it may make sense to use EagerReadDerivation to simplify your query-side models.
- If the write model generates events for all updates, you can structure read models as EventPosters,

- In particular CQRS should only be used on specific portions of a system (a BoundedContext in DDD lingo) and not the system as a whole. In this way of thinking, each Bounded Context needs its own decisions on how it should be modeled.
- So far I see benefits in two directions. Firstly is that a few complex domains may be easier to tackle by using CQRS. I must stress, however, that such suitability for CQRS is very much the minority case. Usually there's enough overlap between the command and query sides that sharing a model is easier. Using CQRS on a domain that doesn't match it will add complexity, thus reducing productivity and increasing risk.
- The other main benefit is in handling high performance applications. CQRS allows you to separate the load from reads and writes allowing you to scale each independently. If your application sees a big disparity between reads and writes this is very handy. Even without that, you can apply different optimization strategies to the two sides. An example of this is using different database access techniques for read and update.
- If your domain isn't suited to CQRS, but you have demanding queries that add complexity or performance problems, remember that you can still use a ReportingDatabase. CQRS uses a separate model for all queries. With a reporting database you still use your main system for most queries, but offload the more demanding ones to the reporting database.

allowing them to be MemoryImages and thus avoiding a lot of database interactions.
- CQRS is suited to complex domains, the kind that also benefit from Domain-Driven Design.

SOLID Principles

S - Single Responsibility principle:
- A class should have one and only one, reason to change.
- When requirements change, this implies that the code has to undergo some reconstruction, meaning that the classes have to be modified. The more responsibilities a class has, the more change requests it will get, and the harder those changes will be to implement. The responsibilities of a class are coupled to each-other, as changes in one of the responsibilities may result in additional changes in order for the other responsibilities to be handled properly by that class.
- What is a responsibility?
    - A responsibility can be defined as a reason for change. Whenever we think that some part of our code is potentially a responsibility, we should consider separating it from the class. Let’s say we are working on a project that helps people become more active in their community, and the system needs to have social media integration. It would be a good idea to separate the social media integration responsibility from the other parts of the system, as we should always be prepared for external changes.

O - Open Closed Principle:
- You should be able to extend a classes behavior, without modifying it.
- Open for extension
    - This ensures that the class behavior can be extended. As requirements change, we should be able to make a class behave in new and different ways, to meet the needs of the new requirements.
- Closed for modification
    - The source code of such a class is set in stone, no one is allowed to make changes to the code.
- How to achieve this?
    - Through abstractions. In order to be able to extend the behavior of a class without changing a single line of code, we need to make abstractions. For example, if we had a system that works with different shapes as classes, we would probably have classes like Circle, Rectangle, etc. In order for a class that depends on one of these classes to implement OCP, we need to introduce a Shape interface/class. Then, wherever we had Dependency Injection, we would inject a Shape instance instead of an instance of a lower-level class. This would give us the luxury

	of adding new shapes without having to change the dependent classes' source code.
L - Liskov substitution principle: ● Derived classes must be substituted for their base classes. ● Solution: ○ We should design by contract. What this means is that each method should have preconditions and postconditions defined. Preconditions must hold true in order for a method to execute, and postconditions must hold true after the execution of a method.	I - Interface segregation principle: ● Client should not be forced to implement interfaces they do not use. ● In other words, it is better to have many smaller interfaces, than fewer, fatter interfaces. ● By breaking down interfaces, we favor Composition instead of Inheritance, and Decoupling over Coupling. We favor composition by separating by roles(responsibilities) and Decoupling by not coupling derivative classes with unneeded responsibilities inside a monolith.
D- Dependency inversion principle ● Depend on abstractions, not on concretions. ● By depending on higher-level abstractions, we can easily change one instance with another instance in order to change the behavior. Dependency Inversion increases the reusability and flexibility of our code.	

Note: Information gathered in this document has been collected from various sources on Internet.
Sources: www.tutorialspoint.com, Others (TBA)