FullStack.Cafe - Kill Your Tech Interview

Q1: What is *Design Patterns* and why anyone should use them? ☆

Topics: Design Patterns

Answer:

Design patterns are a well-described solution to the most commonly encountered problems which occur during software development.

Design pattern represents the best practices evolved over a period of time by experienced software developers. They promote reusability which leads to a more robust and maintainable code.

Q2: What are the *main categories* of Design Patterns? ☆

Topics: Design Patterns

Answer:

The Design patterns can be classified into three main categories:

- Creational Patterns
- Behavioral Patterns
- Functional Patterns

Q3: What is a *pattern*? ☆

Topics: Design Patterns

Answer:

Patterns in programming are like recipes in cooking. They are not ready dishes, but instructions for slicing and dicing products, cooking them, serving them and so forth.

Pattern content As a rule, a pattern description consists of the following:

- a problem that the pattern solves;
- motivation for solving the the problem using the method suggested by the pattern;
- structures of classes comprising the solution;
- an example in one of the programming languages;
- a description of the nuances of pattern implementation in various contexts; relations with other patterns.

Q4: What is *Singleton* pattern? ☆

Topics: Design Patterns

Answer:

Singleton pattern comes under *creational* patterns category and introduces a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Q5: What is *Dependency Injection*? ☆☆

Topics: Design Patterns Dependency Injection

Answer:

Dependency injection makes it easy to create loosely coupled components, which typically means that components consume functionality defined by interfaces without having any first-hand knowledge of which implementation classes are being used.

Dependency injection makes it easier to change the behaviour of an application by changing the components that implement the interfaces that define application features. It also results in components that are easier to isolate for unit testing.

Q6: What is *Inversion of Control*? ☆☆

Topics: Design Patterns

Answer:

Inversion of control is a broad term but for a software developer it's most commonly described as a pattern used for decoupling components and layers in the system.

For example, say your application has a text editor component and you want to provide spell checking. Your standard code would look something like this:

```
public class TextEditor {

    private SpellChecker checker;

    public TextEditor() {
        this.checker = new SpellChecker();
    }
}
```

What we've done here creates a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

```
public class TextEditor {

    private IocSpellChecker checker;

    public TextEditor(IocSpellChecker checker) {
        this.checker = checker;
    }
}
```

You have *inverted control* by handing the responsibility of instantiating the spell checker from the TextEditor class to the caller.

```
SpellChecker sc = new SpellChecker; // dependency
TextEditor textEditor = new TextEditor(sc);
```

Q7: Can we create a *clone* of a singleton object? ☆☆

Topics: Design Patterns

Answer:

Yesl, we can but the purpose of Singleton Object creation is to have single instance serving all requests.

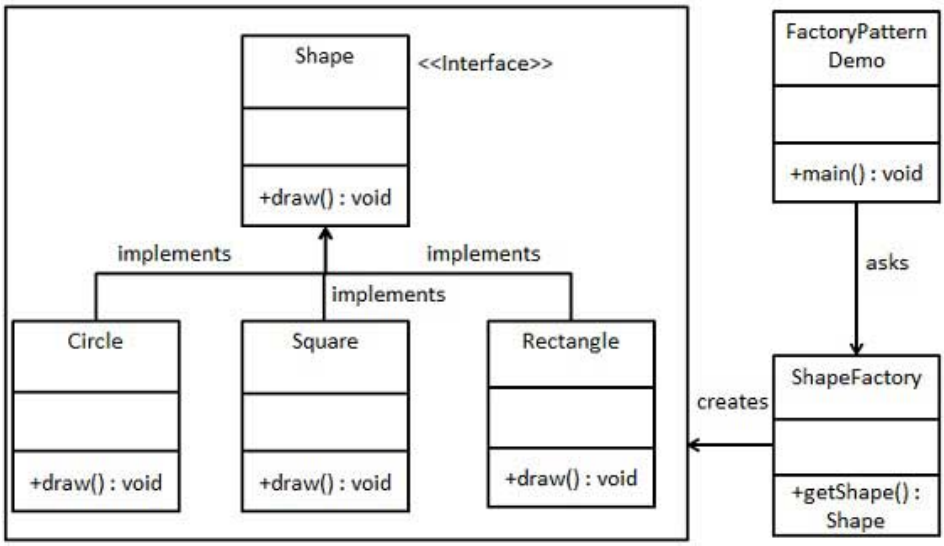
Q8: What is *Factory* pattern? ☆☆

Topics: Design Patterns

Answer:

Factory pattern is one of most used design pattern and comes under *creational* patterns category.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a *common interface*.



Pro's:

- Allows you to hide implementation of an application seam (the core interfaces that make up your application)
- Allows you to easily test the seam of an application (that is to mock/stub) certain parts of your application so you can build and test the other parts
- Allows you to change the design of your application more readily, this is known as loose coupling

Con's

- Makes code more difficult to read as all of your code is behind an abstraction that may in turn hide abstractions.
- Can be classed as an anti-pattern when it is incorrectly used, for example some people use it to wire up a whole application when using an IOC container, instead use Dependency Injection.

Q9: Name *types* of Design Patterns? ☆☆

Topics: Design Patterns

Answer:

Design patterns can be classified in three categories: Creational, Structural and Behavioral patterns.

- Creational Patterns - These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new opreator. This gives program more flexibility in deciding which objects need to be created for a given use case.

- Structural Patterns - These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

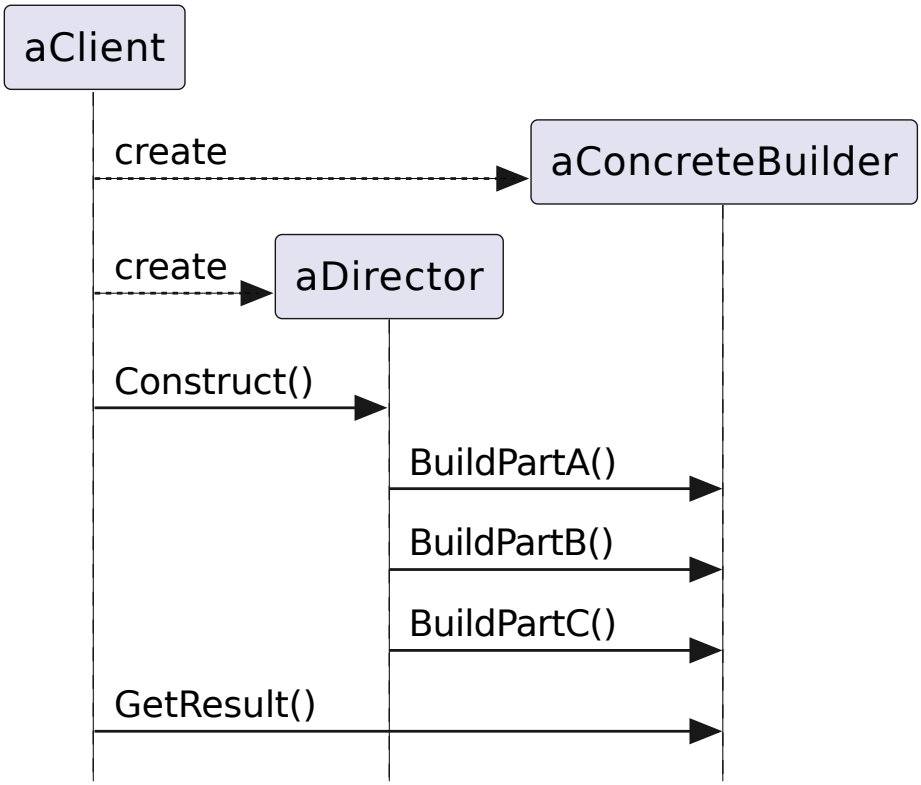
- Behavioral Patterns - These design patterns are specifically concerned with communication between objects.

Q10: What is *Builder* pattern? ☆☆

Topics: Design Patterns

Answer:

Builder pattern builds a complex object using simple objects and using a step by step approach. This builder is independent of other objects.



The Director class is optional and is used to make sure that the building steps are executed in the *right order* with the right data by the right builder. It's about validation and delegation.

Builder/Director pattern's steps invocations could be semantically presented by *method chaining* or so called *Fluent Interface* syntax.

```
Pizza pizza = new Pizza.Builder()
                       .cheese(true)
                       .pepperoni(true)
                       .bacon(true)
                       .build();
```

Q11: What is *Filter* pattern? ☆☆

Topics: Design Patterns

Answer:

Filter pattern or **Criteria pattern** is a design pattern that enables developers to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations. This type of design pattern comes under *structural* pattern as this pattern combines multiple criteria to obtain single criteria.

Filter design pattern is useful where you want to add filters dynamically or you are implementing multiple functionalities and most of them require different filter criteria to filter something. In that case instead of hard coding the filters inside the functionalities, you can create filter criteria and re-use it wherever required.

```
List<Laptop> laptops = LaptopFactory.manufactureInBulk();
AndCriteria searchCriteria = new AndCriteria(
  new HardDisk250GBFilter(),
  new MacintoshFilter(),
  new I5ProcessorFilter());
List<Laptop> filteredLaptops = searchCriteria.meets(laptops);
```

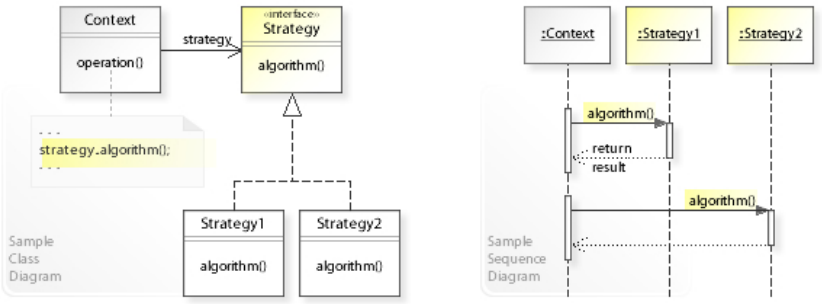
Q12: What is *Strategy* pattern? ☆☆

Topics: Design Patterns

Answer:

In **Strategy pattern**, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under *behavior* pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

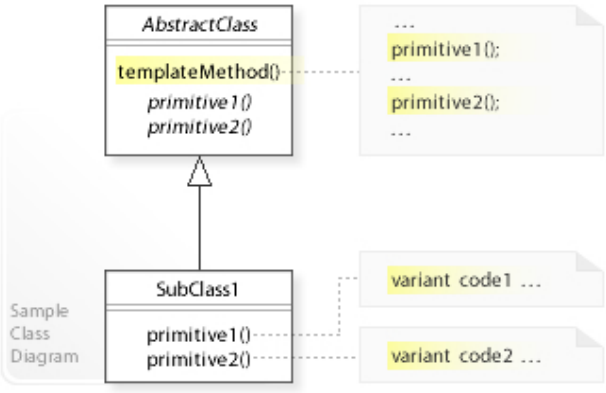


Q13: What is *Template* pattern? ☆☆

Topics: Design Patterns

Answer:

In **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. This pattern comes under *behavior* pattern category.



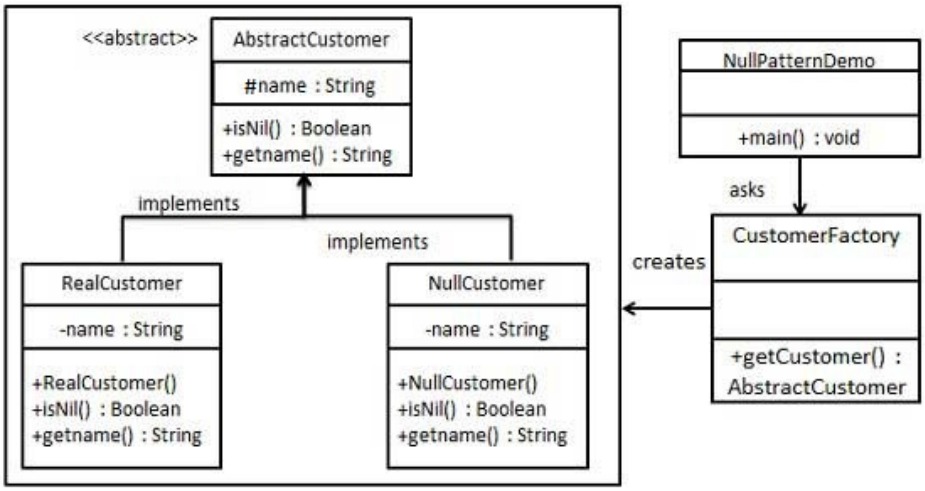
Q14: What is *Null Object* pattern? ☆☆

Topics: Design Patterns

Answer:

In **Null Object pattern**, a null object replaces check of NULL object instance. Instead of putting if check for a null value, Null Object reflects a do nothing relationship. Such Null object can also be used to provide default behaviour in case data is not available.

In Null Object pattern, we create an abstract class specifying various operations to be done, concrete classes extending this class and a null object class providing do nothing implementation of this class and will be used seamlessly where we need to check null value.

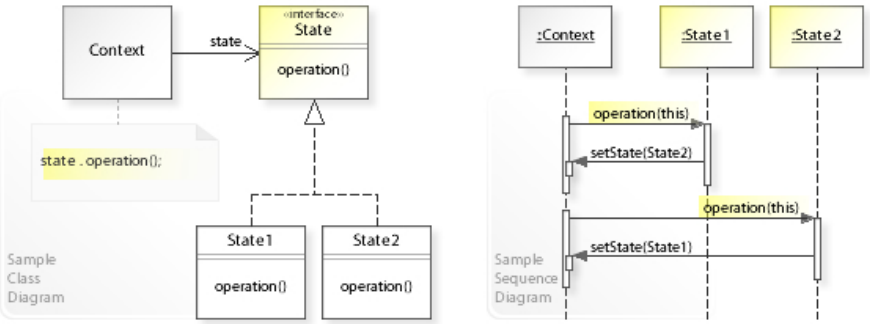


Q15: What is *State* pattern? ☆☆

Topics: Design Patterns

Answer:

In **State pattern** a class behavior changes based on its state. This type of design pattern comes under *behavior* pattern. In State pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.

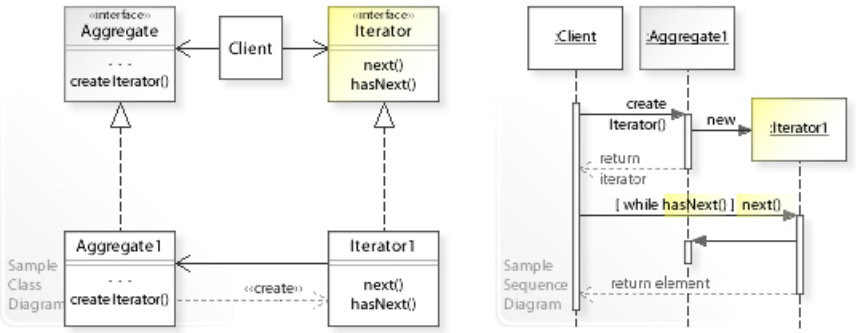


Q16: What is *Iterator* pattern? ☆☆

Topics: Design Patterns

Answer:

Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation. Iterator pattern falls under *behavioral* pattern category.



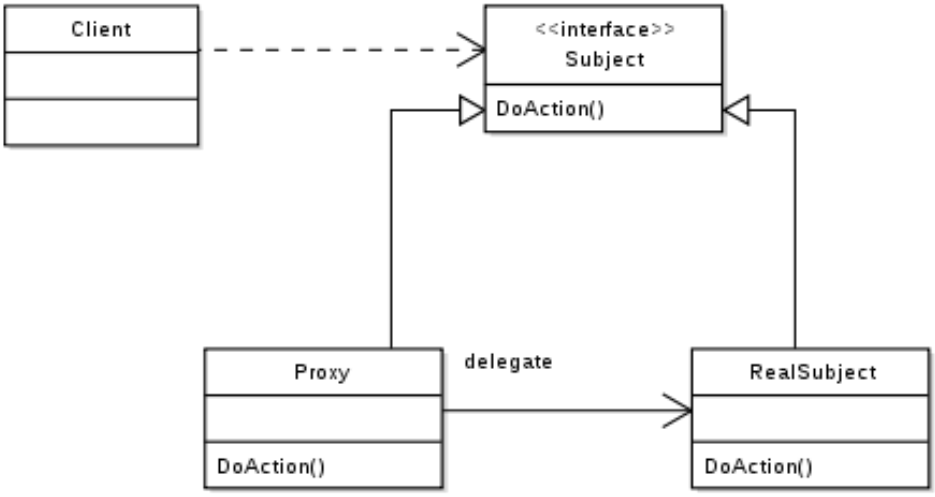
Q17: What is *Proxy* pattern? ☆☆

Topics: Design Patterns

Answer:

In **proxy pattern**, a class represents functionality of another class. This type of design pattern comes under *structural* pattern.

In proxy pattern, we create object having original object to interface its functionality to outer world.



Q18: What are some benefits of *Repository Pattern*? ☆☆

Topics: Design Patterns

Answer:

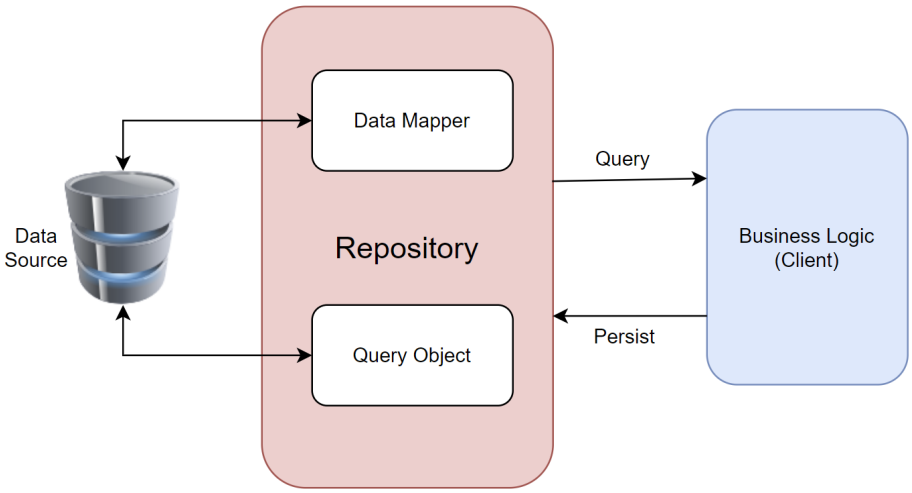
A **Repository Pattern** allows all of your code to *use objects without having to know how the objects are persisted*. All of the knowledge of persistence, including mapping from tables to objects, is safely contained in the repository.

Under the covers:

- for reading, it creates the query satisfying the supplied criteria and returns the result set
- for writing, it issues the commands necessary to make the underlying persistence engine (e.g. an SQL database) save the data

Very often, you will find SQL queries scattered in the codebase and when you come to add a column to a table you have to search code files to try and find usages of a table. The impact of the change is far-reaching.

With the repository pattern, you would only need to change one object and one repository. The impact is very small. There are other benefits too, for example, if you were using MySQL and wanted to switch to SQL Server.



Q19: Why would you want to use a *Repository Pattern* with an ORM? ☆☆

Topics: Design Patterns

Answer:

- The **Repository** abstract the access to **all storage concerns**. The Repository allows the rest of the application to ignore persistence details. Repositories deal with Domain/Business objects.
- The **ORM** abstract access to a **specific RDBMS**. An ORM handles db objects. The ORM is an implementation detail of the Repository.

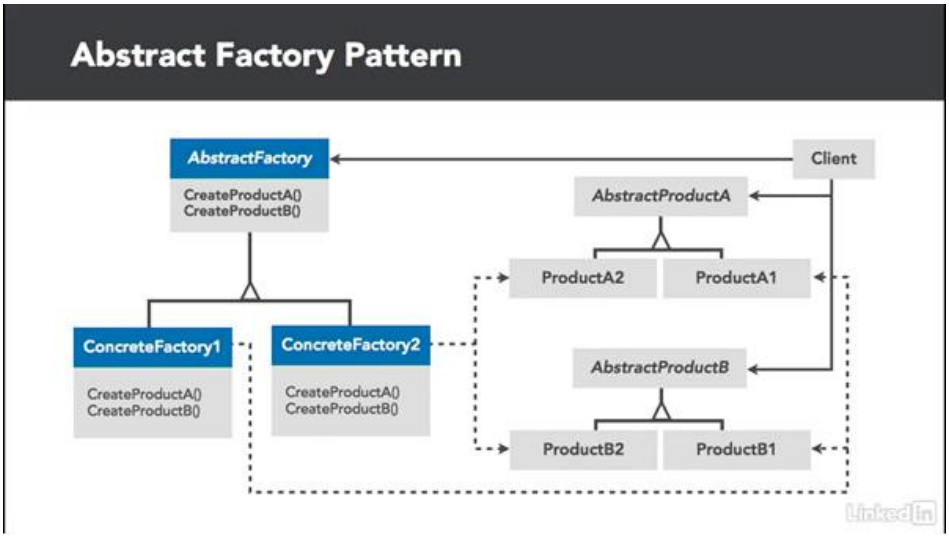
Q20: What is *Abstract Factory* pattern? ☆☆☆

Topics: Design Patterns

Answer:

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.



Pros:

- Follows the Open/Closed Principle.
- Allows building families of product objects and guarantees their compatibility.
- Avoids tight coupling between concrete products and code that uses them.
- Divides responsibilities between multiple classes.

Cons:

- Increases overall code complexity by creating multiple additional classes.

FullStack.Cafe - Kill Your Tech Interview

Q1: What is *Unit Of Work*? ☆☆☆

Topics: ADO.NET OOP Design Patterns

Answer:

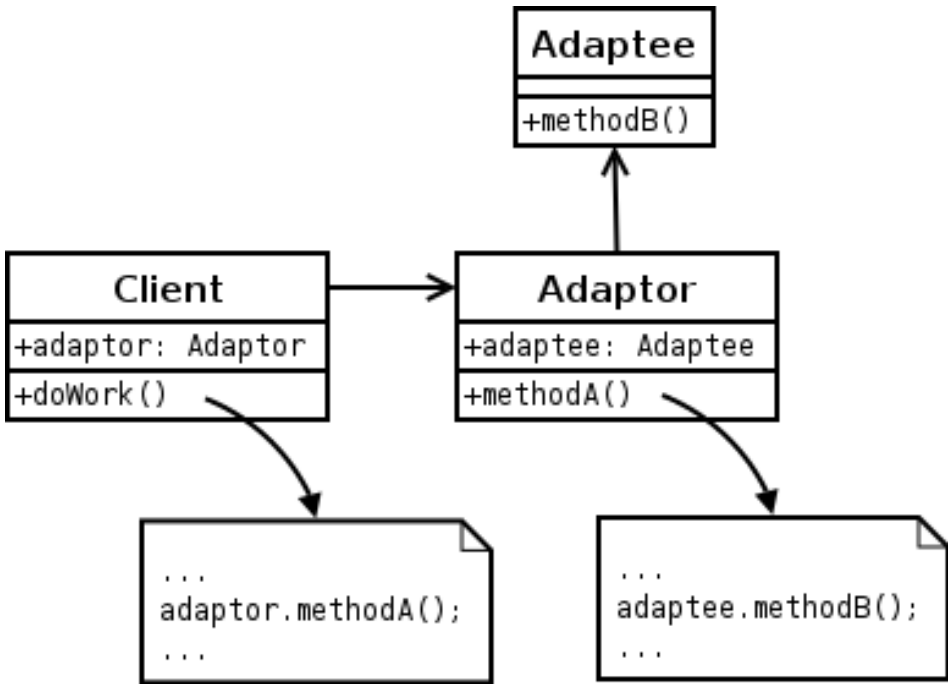
Unit of Work is referred to as a single transaction that involves multiple operations of `insert/update/delete` and so on kinds. To say it in simple words, it means that for a specific user action (say registration on a website), all the transactions like insert/update/delete and so on are done in one single transaction, rather than doing multiple database transactions.

Q2: What is *Adapter* Pattern? ☆☆☆

Topics: Design Patterns

Answer:

Adapter pattern works as a bridge between two incompatible interfaces. This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces (adaptees).



A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

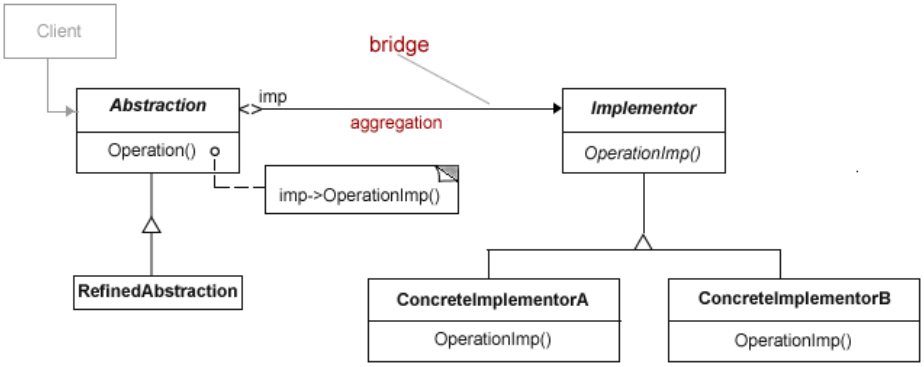
Q3: What is *Bridge* pattern? ☆☆☆

Topics: Design Patterns

Answer:

Bridge pattern is used when we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under *structural* pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.

The bridge pattern is useful when both the class and what it does vary often. The class itself can be thought of as the abstraction and what the class can do as the implementation. The bridge pattern can also be thought of as two layers of abstraction.



This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

The example of bridge pattern implementation is when:

```
                  ----Shape---
                 /            \
         Rectangle             Circle
        /         \           /      \
BlueRectangle  RedRectangle BlueCircle RedCircle
```

refactored to:

```
          ----Shape---                      Color
         /            \                    /   \
Rectangle(Color)   Circle(Color)          Blue   Red
```

or in general when:

```
        A
      /    \
    Aa      Ab
   / \     /  \
 Aa1 Aa2  Ab1 Ab2
```

refactored to:

```
     A         N
   /    \     / \
Aa(N) Ab(N)  1   2
```

Q4: What does *program to interfaces, not implementations* mean?
☆☆☆

Topics: Design Patterns Software Architecture

Answer:

Coding against interface means, the client code always holds an Interface object which is supplied by a *factory*.

Any instance returned by the factory would be of type Interface which any factory candidate class must have implemented. This way the client program is not worried about implementation and the interface signature determines what all operations can be done.

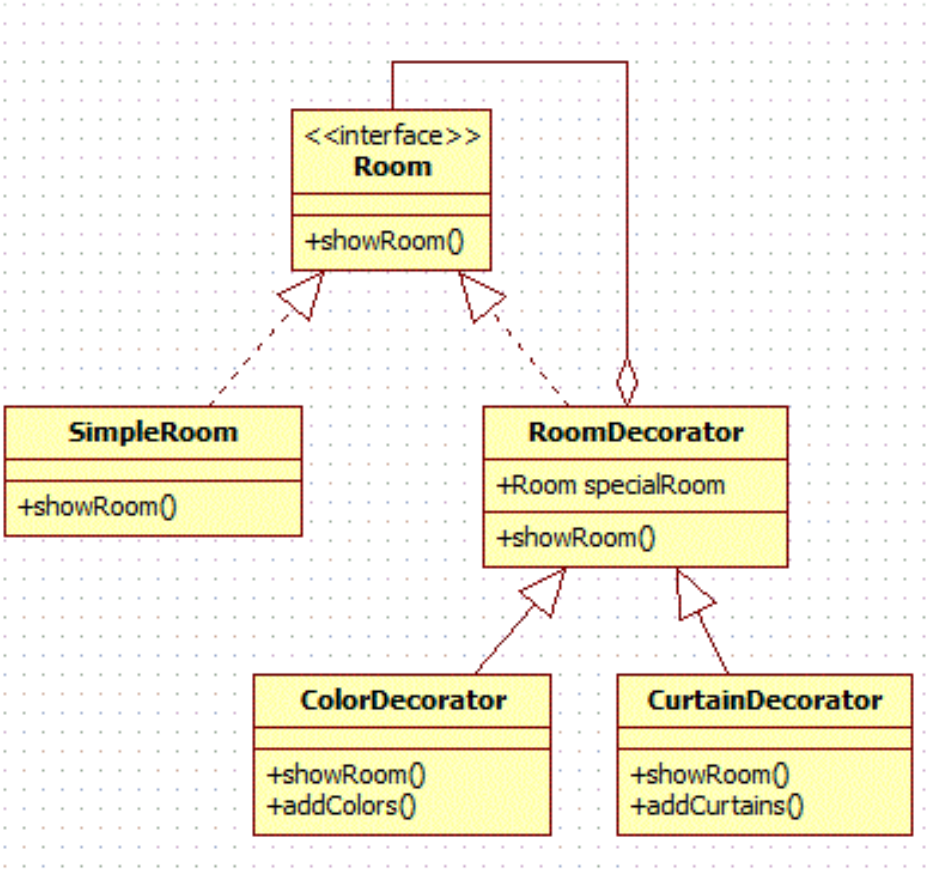
This approach can be used to change the behavior of a program at run-time. It also helps you to write far better programs from the maintenance point of view.

Q5: What is *Decorator* pattern? ☆☆☆

Topics: Design Patterns

Answer:

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under *structural pattern* as this pattern acts as a wrapper to existing class.



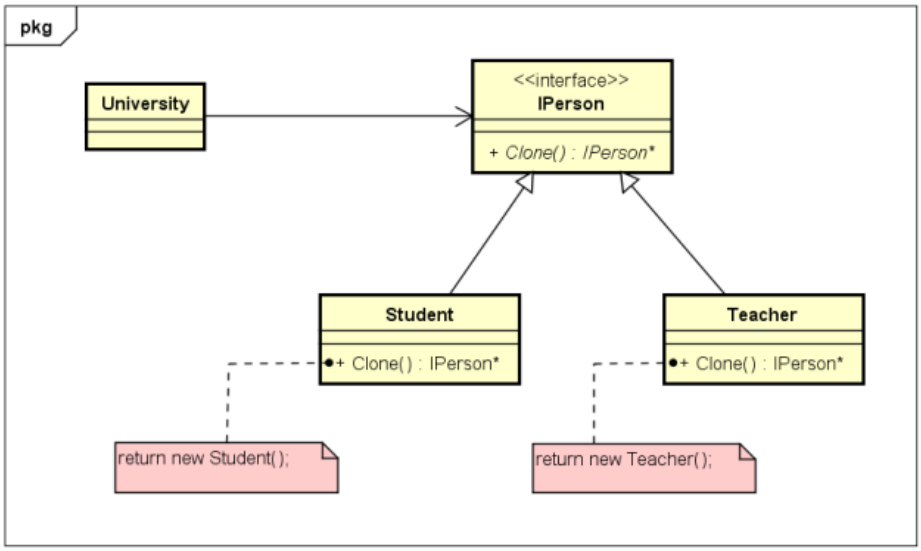
This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

Q6: What is *Prototype* pattern? ☆☆☆

Topics: Design Patterns

Answer:

Prototype pattern refers to creating duplicate object while keeping performance in mind. This pattern involves implementing a prototype interface which tells to create a clone of the current object.



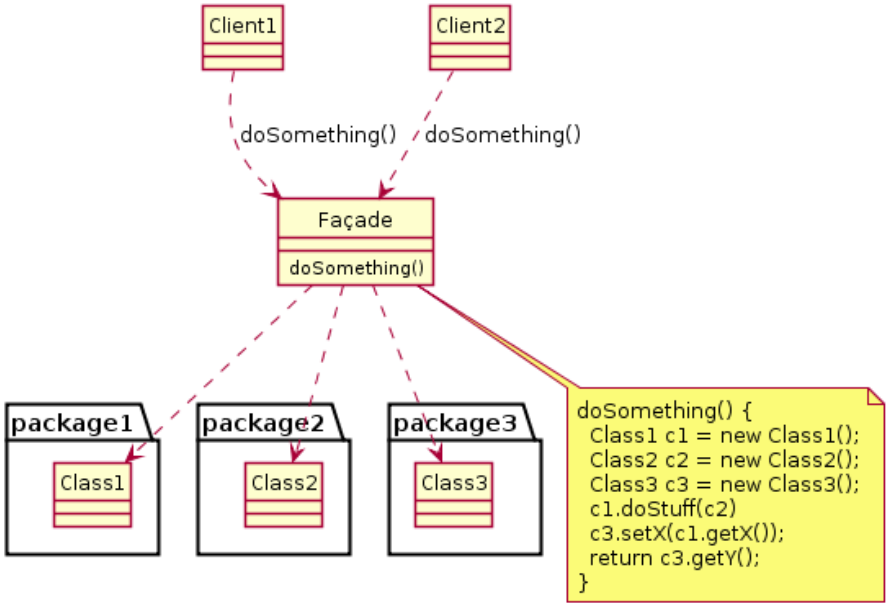
The Prototype pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.

Q7: What is *Facade* pattern? ☆☆☆

Topics: Design Patterns

Answer:

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under *structural pattern* as this pattern adds an interface to existing system to hide its complexities.



This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

Q8: Can you give any good explanation what is the difference between *Proxy* and *Decorator*? ☆☆☆

Topics: Design Patterns

Answer:

Decorator Pattern focuses on dynamically adding functions to an object, while **Proxy Pattern** focuses on controlling access to an object.

Q9: What are the difference between a *Static* class and a *Singleton* class? ☆☆☆

Topics: Design Patterns

Answer:

Following are the differences between a static class and a singleton class.

- A static class can not be a top level class and can not implement interfaces where a singleton class can.
- All members of a static class are static but for a Singleton class it is not a requirement.
- A static class get initialized when it is loaded so it can not be lazily loaded where a singleton class can be lazily loaded.
- A static class object is stored in stack whereas singlton class object is stored in heap memory space.

Q10: When should I use *Composite* design pattern? ☆☆☆

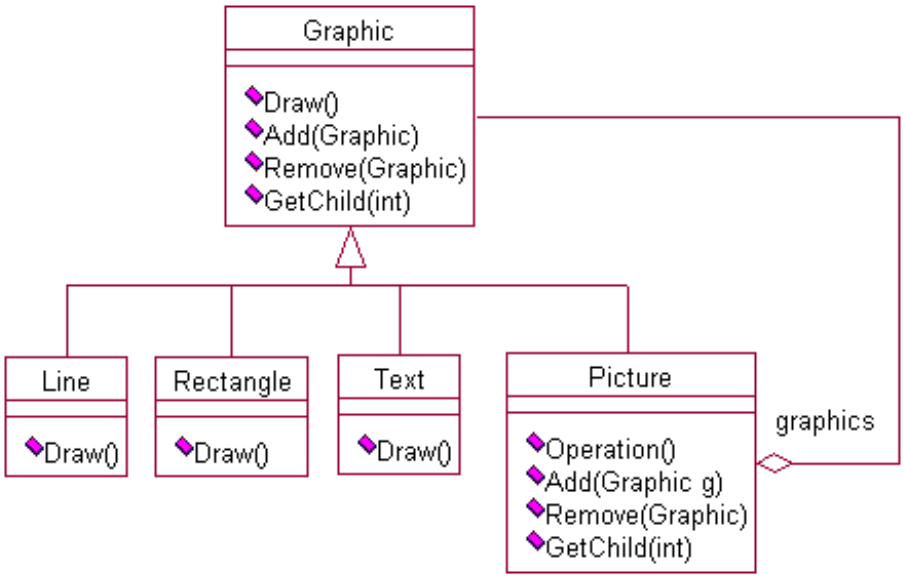
Topics: Design Patterns

Answer:

Use the **Composite Pattern** when

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

A common usage is a display system of graphic windows which can contain other windows and graphic elements such as images, text. The composite can be composed at run-time, and the client code can manipulate all the elements without concern for which type it is for common operations such as drawing. Another example is directories contain entries, each of which could be a directory.

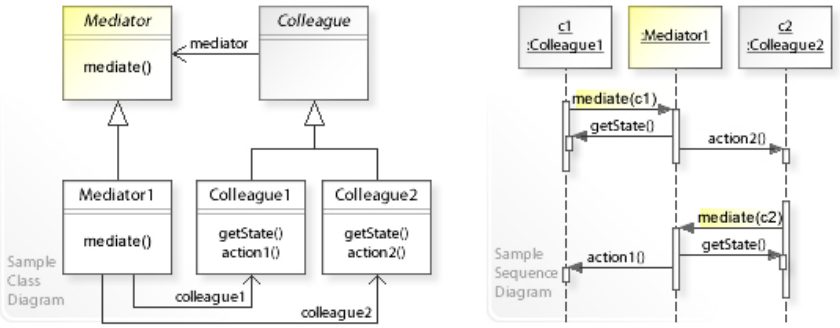


Q11: What is *Mediator* pattern? ☆☆☆

Topics: Design Patterns

Answer:

Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintenance of the code by loose coupling. Mediator pattern falls under *behavioral* pattern category.



Q12: What is *Observer* pattern? ☆☆☆

Topics: Design Patterns

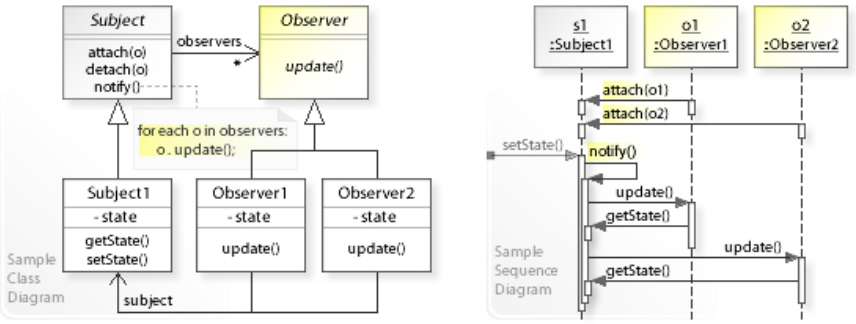
Answer:

Observer pattern (also known as *Publish-Subscribe 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. Observer pattern falls under *behavioral* pattern category.

An object with a one-to-many relationship with other objects who are interested in its state is called the *subject* or *publisher*. The *observers* are notified whenever the state of the *subject* changes and can act accordingly. The *subject* can have any number of dependent *observers* which it notifies, and any number of *observers* can subscribe to the *subject* to receive such notifications.

Observer pattern uses two actor classes:

- The Observer (os Subscriber) abstract class provides an `update()` method which will be called by the subject to notify it of the subject's state change.
- The Subject (or Publisher) class is also an abstract class and defines four primary methods: `attach()`, `detach()`, `setState()`, and `notify()`

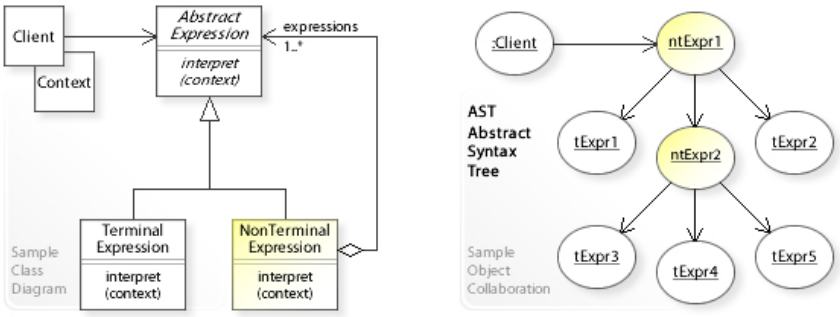


Q13: What is *Interpreter* pattern? ☆☆☆

Topics: Design Patterns

Answer:

Interpreter pattern provides a way to evaluate language grammar or expression. This type of pattern comes under *behavioral* pattern. This pattern involves implementing an expression interface which tells to interpret a particular context.



This class diagram means that an `AbstractExpression` is either a `TerminalExpression` or a `NonTerminalExpression`. If its a `NonTerminalExpression`, it is itself an aggregation of one or several `AbstractExpression`.

Any mechanism for interpreting formal languages suites this pattern perfectly, it can be anything: from a simple calculator to a C# parser.

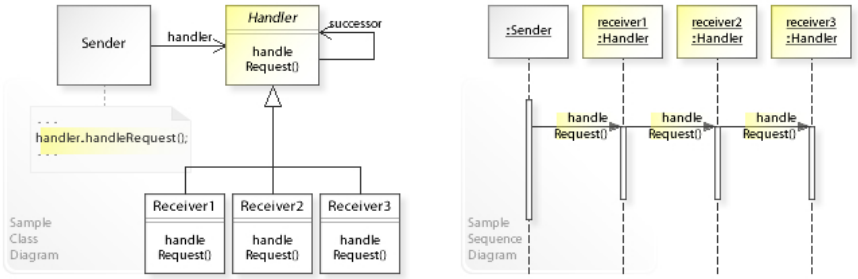
Q14: What is the *Chain of Responsibility* pattern? ☆☆☆

Topics: Design Patterns

Answer:

As the name suggests, the **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. This pattern comes under *behavioural* patterns.

In this pattern, normally each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on. The chain of responsibility is an object oriented version of the `if ... else if ... else if ..... else ... endif` idiom, with the benefit that the condition–action blocks can be dynamically rearranged and reconfigured at runtime.



Q15: What is *Memento* pattern? ☆☆☆

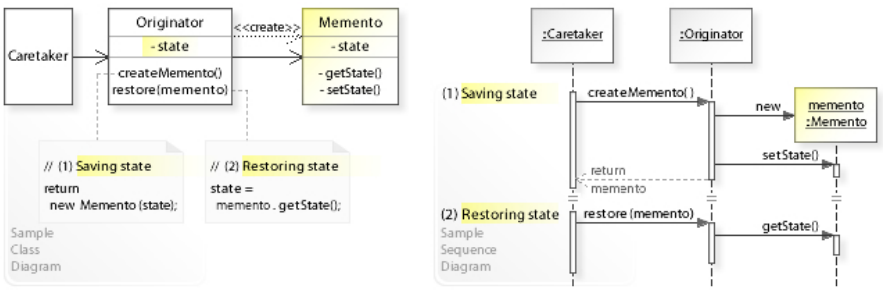
Topics: Design Patterns

Answer:

Memento pattern is used to restore state of an object to a previous state. Memento pattern falls under *behavioral* pattern category.

Memento pattern uses three actor classes:

- *Memento* contains state of an object to be restored.
- *Originator* creates and stores states in Memento objects and
- *Caretaker* object is responsible to restore object state from Memento.

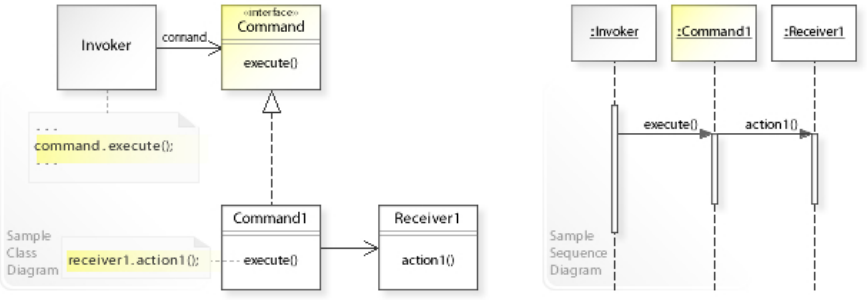


Q16: What is *Command* pattern? ☆☆☆

Topics: Design Patterns

Answer:

Command pattern is a data driven design pattern and falls under *behavioural* pattern category. A request is wrapped under an object as *command* and passed to *invoker* object. *Invoker* object looks for the appropriate object which can handle this command and passes the command to the corresponding *receiver* which executes the command.



Q17: What are some reasons to use *Repository Pattern*? ☆☆☆

Topics: Design Patterns

Answer:

Here are some reasons to use **Repository Pattern**:

- You have a single place to make changes to your data access
- You have a single place responsible for a set of tables (usually)
- It is easy to replace a repository with a fake implementation for testing - so you don't need to have a database available for your unit tests

Q18: What is an *Aggregate Root* in the context of *Repository Pattern*? ☆☆☆

Topics: Design Patterns

Answer:

In the context of the repository pattern, **aggregate roots** are the only objects your client code loads from the repository.

The repository encapsulates access to child objects - from a caller's perspective it automatically loads them, either at the same time the root is loaded or when they're actually needed (as with lazy loading).

For example, you might have an `Order` object which encapsulates operations on multiple `LineItem` objects. Your client code would never load the `LineItem` objects directly, just the `Order` that contains them, which would be the aggregate root for that part of your domain.

Q19: Is *Unit Of Work* equals *Transaction*? Or it is more than that? ☆☆☆

Topics: Design Patterns Software Architecture

Answer:

A `UnitOfWork` is a business transaction. Not necessarily a technical transaction (db transaction) but often tied to technical transactions.

In the Enterprise Application Patterns it is defined as

> Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

A database transaction with a number of SQL statements in between is arguably also a Unit of Work. However, the key difference, is that the unit of work, as defined in the pattern, has abstracted that level of detail (how changes are written and the storage type) to an object level.

Q20: When should I use *Active Record* vs *Repository Pattern*? ☆☆☆

Topics: Design Patterns

Answer:

The **Repository** and DAO patterns *externalize* persistence concerns while **Active Record** *internalizes* them. I've actually seen some implementations where active record classes were injected with a repository instance that provided their persistence concerns internally.

The biggest reason against using the Active Record pattern is simple, your domain objects shouldn't care how (or even if) they are persisted. The repository pattern provides persistence ignorance to your domain objects by externalizing persistence concerns and providing it as an external service.

Active Record is a good choice for domain logic that isn't too complex, such as creates, reads, updates, and deletes. Derivations and validations based on a single record work well in this structure. Active Record has the primary advantage of simplicity. It's easy to build Active Records, and they are easy to understand. Their primary problem is that they work well only if the Active Record objects correspond directly to the database tables: an isomorphic schema.

If your business logic is complex, you'll soon want to use your object's direct relationships, collections, inheritance, and so forth. These don't map easily onto Active Record and Repository Pattern is more relevant.

FullStack.Cafe - Kill Your Tech Interview

Q1: What are the drawbacks to the *ActiveRecord* pattern? ☆☆☆

Topics: Design Patterns

Answer:

The main drawback is your "entities" are aware of their own *persistence* which leads to a lot of other bad design decisions. Having your objects know about their persistence means you need to do things like:

- easily have database connections available everywhere. This typically leads to nasty hardcoding or some sort of static connection that gets hit from everywhere.
- your objects tend to look more like SQL than objects.
- hard to do anything in the app disconnected because database is so ingrained.

The other issues is that most active record toolkits basically map 1 to 1 to table fields with zero layers of indirection. This works on small scales but falls apart when you have trickier problems to solve.

Q2: In OOP, what is the difference between the *Repository Pattern* and a *Service Layer*? ☆☆☆

Topics: ASP.NET ASP.NET MVC Design Patterns Software Architecture ASP.NET Web API

Answer:

- **Repository Layer** gives you additional level of abstraction over data access.
- **Service Layer** exposes business logic, which *uses* repository or a set of repositories as *UnitOfWork*

In ASP.NET MVC + EF + SQL Server, I have this flow of communication:

- **Views <- Controllers -> Service layer -> Repository layer -> EF -> SQL Server**
- **Service layer -> Repository layer -> EF** This part operates on models.
- **Views <- Controllers -> Service layer** This part operates on view models.

Q3: What are some *advantages* of using Dependency Injection ☆☆☆

Topics: Design Patterns Dependency Injection

Answer:

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. When using dependency injection, objects are given their dependencies *at run time rather than compile time (car manufacturing time)*.

- It allows your code to be more loosely coupled because classes do not have hard-coded dependencies
- Decoupling the creation of an object (in other words, separate usage from the creation of an object)
- Making isolation in unit testing possible/easy. It is harder to isolate components in unit testing without dependency injection.

- **Explicitly** defining dependencies of a class
- Facilitating good design like the single responsibility principle (SRP) for example
- Promotes *Code to an interface, not to implementation* principle
- Enabling switching/ability to replace dependencies/implementations quickly (`DbLogger` instead of `ConsoleLogger` for example)

Q4: What is the *Command and Query Responsibility Segregation (CQRS)* Pattern? ☆☆☆

Topics: DDD Software Architecture Design Patterns

Answer:

In traditional architectures, the same data model is used to query and update a database. That's simple and works well for basic CRUD operations.

CQRS stands for **Command and Query Responsibility Segregation**, a pattern that separates read and update operations for a data store. CQRS separates *reads* and *writes* into different models, using **commands** to update data, and **queries** to read data.

- Commands should be task-based, rather than data-centric. ("Book hotel room", not "set ReservationStatus to Reserved").
- Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
- Queries never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

Q5: Name some benefits of *CQRS Pattern* ☆☆☆

Topics: DDD Software Architecture Design Patterns

Answer:

Benefits of CQRS include:

- **Independent scaling**. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.
- **Optimized data schemas**. The read side can use a schema that is optimized for queries, while the write side uses a schema that is optimized for updates.
- **Security**. It's easier to ensure that only the right domain entities are performing writes on the data.
- **Separation of concerns**. Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.
- **Simpler queries**. By storing a materialized view in the read database, the application can avoid complex joins when querying.

Q6: Describe what is the *Event Sourcing Pattern* ☆☆☆

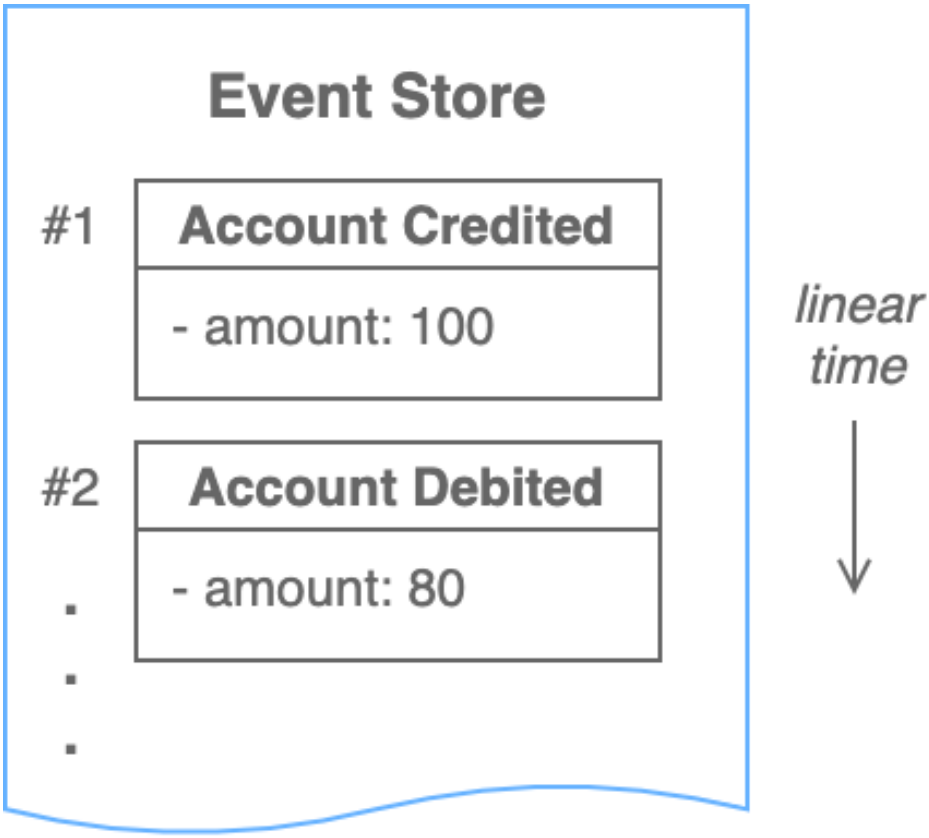
Topics: DDD Software Architecture Design Patterns

Answer:

Event Sourcing is a pattern for the recording of state in a *non-destructive* way. **Each change of state is appended to a log**. Because the changes are non-destructive, we preserve the ability to answer queries about the state of the object at any point in its life cycle.

- Event sourcing enables *traceability* of changes.
- Event sourcing enables *audit logs* without any additional effort.
- Event sourcing makes it possible to *reinterpret* the past.
- Event sourcing *reduces the conflict* potential of simultaneously occurring changes.
- Event sourcing enables *easy versioning* of business logic.

Event Sourcing is not necessary for **CQRS**. You can combine Event Sourcing and CQRS. This kind of combination can lead us to a new type of CQRS. It involves modelling the state changes made by applications as an immutable sequence or log of events.



Q7: Could you explain some benefits of *Repository Pattern*? ☆☆☆☆

Topics: ADO.NET OOP Design Patterns

Answer:

Very often, you will find SQL queries scattered in the codebase and when you come to add a column to a table you have to search code files to try and find usages of a table. The impact of the change is far-reaching. **With the repository pattern, you would only need to change one object and one repository**. The impact is very small.

Basically, **repository** hides the details of how exactly the data is being fetched/persisted from/to the database. Under the covers:

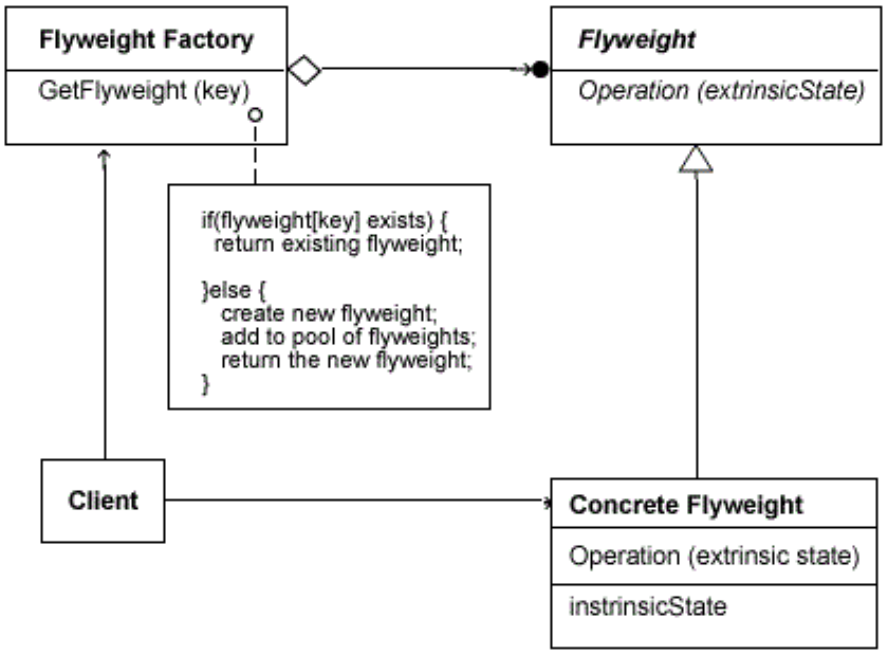
- for reading, it creates the query satisfying the supplied criteria and returns the result set
- for writing, it issues the commands necessary to make the underlying *persistence* engine (e.g. an SQL database) save the data

Q8: What is *Flyweight* pattern? ☆☆☆☆

Topics: Design Patterns

Answer:

Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance. This type of design pattern comes under *structural pattern* as this pattern provides ways to decrease object count thus improving the object structure of application.



Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found.

Q9: When would you use the *Builder* Pattern? Why not just use a *Factory* Pattern? ☆☆☆☆

Topics: Design Patterns

Answer:

The *builder pattern* is a good choice when designing classes whose constructors or static factories would have more than a handful of parameters.

Consider a restaurant. The creation of "today's meal" is a factory pattern, because you tell the kitchen "get me today's meal" and the kitchen (factory) decides what object to generate, based on hidden criteria.

The builder appears if you order a custom pizza. In this case, the waiter tells the chef (builder) "I need a pizza; add cheese, onions and bacon to it!" Thus, the builder exposes the attributes the generated object should have, but hides how to set them.

Q10: How is *Bridge* pattern is different from *Adapter* pattern? ☆☆☆☆

Topics: Design Patterns

Answer:

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.

Q11: Why would I ever use a *Chain of Responsibility* over a *Decorator*? ☆☆☆☆

Topics: Design Patterns

Answer:

The key difference is that a **Decorator** adds new behaviour that in effect widens the original interface. It is similar to how normal extension can add methods except the "subclass" is only coupled by a reference which means that any "superclass" can be used.

The **Chain of Responsibility** pattern can modify an existing behaviour which is similar to overriding an existing method using inheritance. You can choose to call `super.xxx()` to continue up the "chain" or handle the message yourself.

Q12: What is the difference between *Strategy* design pattern and *State* design pattern? ☆☆☆☆

Topics: Design Patterns

Answer:

The difference simply lies in that they solve different problems:

- The **State pattern** deals with **what** (state or type) an object is (in) -- it encapsulates state-dependent behavior, whereas
- the **Strategy pattern** deals with **how** an object performs a certain task -- it encapsulates an algorithm.

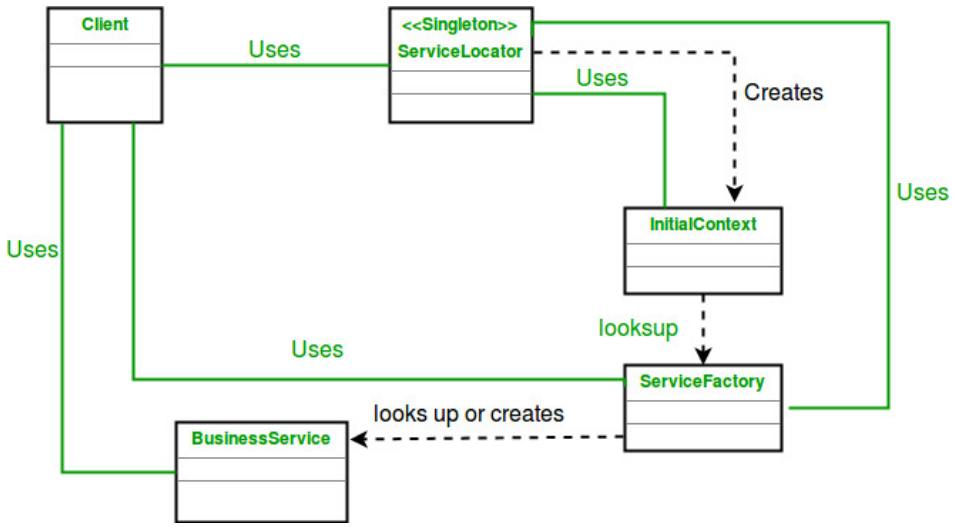
The constructs for achieving these different goals are however very similar; both patterns are examples of composition with delegation.

Q13: Explain usage of *Service Locator* Pattern ☆☆☆☆

Topics: Design Patterns

Answer:

The **service locator** pattern is a design pattern used in software development to encapsulate the processes involved in obtaining a service with a strong abstraction layer. This pattern uses a central registry known as the “service locator” which on request returns the information necessary to perform a certain task. The ServiceLocator is responsible for returning instances of services when they are requested for by the service consumers or the service clients.



- **Service Locator**: The Service Locator abstracts the API lookup services, vendor dependencies, lookup complexities, and business object creation, and provides a simple interface to clients. This reduces the client's complexity. In addition, the same client or other clients can reuse the Service Locator.
- **InitialContext**: The InitialContext object is the start point in the lookup and creation process. Service providers provide the context object, which varies depending on the type of business object provided by the Service Locator's lookup and creation service.
- **ServiceFactory**: The ServiceFactory object represents an object that provides life cycle management for the BusinessService objects. The ServiceFactory object for enterprise beans is an EJBHome object.
- **BusinessService**: The BusinessService is a role that is fulfilled by the service the client is seeking to access. The BusinessService object is created or looked up or removed by the ServiceFactory. The BusinessService object in the context of an EJB application is an enterprise bean.

Q14: Explain what is *Composition* over *Inheritance*? ☆☆☆☆

Topics: Design Patterns

Answer:

Composition over inheritance (or composite reuse principle) in object-oriented programming (OOP) is the principle that classes should achieve *polymorphic behavior* and code reuse by their composition (by containing instances of other classes that implement the desired functionality) rather than inheritance from a base or parent class. It is more natural to build business-domain classes out of various components than trying to find commonality between them and creating a family tree.

One common drawback of using composition instead of inheritance is that methods being provided by individual components may have to be implemented in the derived type, even if they are only forwarding methods.

Some languages, notably *Go*, use type composition exclusively.

Q15: What will you choose: *Repository Pattern* or "smart" business objects? ☆☆☆☆

Topics: Design Patterns Software Architecture

Problem:

Some folks use the "repository pattern" which uses a repository that knows how to fetch, insert, update and delete objects. Those objects are rather "dumb" in that they don't necessarily contain a whole lot of logic - e.g.

they're more or less data-transfer objects.

The other camp uses what I call "smart" business objects that know how to load themselves, and they typically have a Save(), possibly Update() or even Delete() method. Here you really don't need any repository - the objects themselves know how to load and save themselves.

What are your thoughts?

Solution:

I use the **Repository Pattern** because of:

- the *Single Responsibility Principle*. I don't want each individual object having to know how to save, update, delete itself when this can be handled by one single generic repository,
- It also makes unit testing simpler as well,
- data transfer objects (DTO) are more flexible. You can use them everywhere, with no dependency on frameworks, layers, etc.
- the repository pattern doesn't necessarily lead to dumb objects. If the objects have no logic outside Save/Update, you're probably doing too much outside the object. Ideally, you should never use properties to get data from your object, compute things, and put data back in the object. This is a break of encapsulation.

Q16: Is *Repository Pattern* as same as *Active Record Pattern*? ☆☆☆☆

Topics: Design Patterns Software Architecture

Answer:

Big difference between *Active Record* and *Repository patterns* is in my opinion the owner of the link between entity instance and underlying storage:

- **Active Record Pattern** defines An **object that wraps a row in a database** table or view, encapsulates the data access, and adds domain logic to that data. In Active Record, entity instance knows how and where to persist itself (this is what "active" means in my mind). That's why you can just call `user.save()` and it persists itself.

- **In the Repository pattern** all of the **data access is put in a separate class and is accessed via instance methods**. To me, just doing this is beneficial, since data access is now encapsulated in a separate class, leaving the business object to get on with business. This should stop the unfortunate mixing of data access and business logic you tend to get with Active Record. In Repository pattern, entity is more or less dumb POJO, it's the repository that manages its lifecycle. If you create a new instance of the entity, it's not magically persisted, you need to tell the repository to persist it.

Q17: How should I be *grouping* my Repositories when using Repository Pattern? ☆☆☆☆

Topics: Design Patterns Software Architecture

Answer:

One common mistake when using Repository Pattern is to think that table relates to repository 1:1.

Instead, repository should be per *Aggregate Root* and not a *table*. It means - if an entity shouldn't live alone (i.e. - if you have a `Registrant` that participates in a particular `Registration`) - it's just an entity and it should be updated/created/retrieved through a repository of *Aggregate Root* it belongs.

In many cases, this technique of reducing the count of repositories. To avoid that simplification you can cascade repositories through IoC like in this example:

```
var registrationService = new RegistrationService(new RegistrationRepository(),
      new LicenseRepository(), new GodOnlyKnowsWhatElseThatServiceNeeds());
```

Q18: Why shouldn't I use the *Repository Pattern* with Entity Framework? ☆☆☆☆

Topics: .NET Core Design Patterns Entity Framework

Answer:

The single best reason to not use the repository pattern with Entity Framework? Entity Framework *already* implements a repository pattern. `DbContext` is your UoW (Unit of Work) and each `DbSet` is the repository. Implementing another layer on top of this is not only redundant but makes maintenance harder.

In the case of the repository pattern, the purpose is to abstract away the low-level database querying logic. In the old days of actually writing SQL statements in your code, the repository pattern was a way to move that SQL out of individual methods scattered throughout your codebase and localize it in one place. Having an ORM like Entity Framework, NHibernate, etc. is a *replacement* for this code abstraction, and as such, negates the need for the pattern.

However, it's not a bad idea to create an abstraction on top of your ORM, just not anything as complex as the UoW/repository. I'd go with a service pattern, where you construct an API that your application can use without knowing or caring whether the data is coming from Entity Framework, NHibernate, or a Web API. This is much simpler, as you merely add methods to your service class to return the data your application needs. The key difference here is that a service is a thinner layer and is geared towards returning fully-baked data, rather than something that you continue to query into, like with a repository.

Q19: What is relationship between *Repository* and *Unit of Work*? ☆☆☆☆

Topics: Design Patterns Software Architecture Entity Framework

Answer:

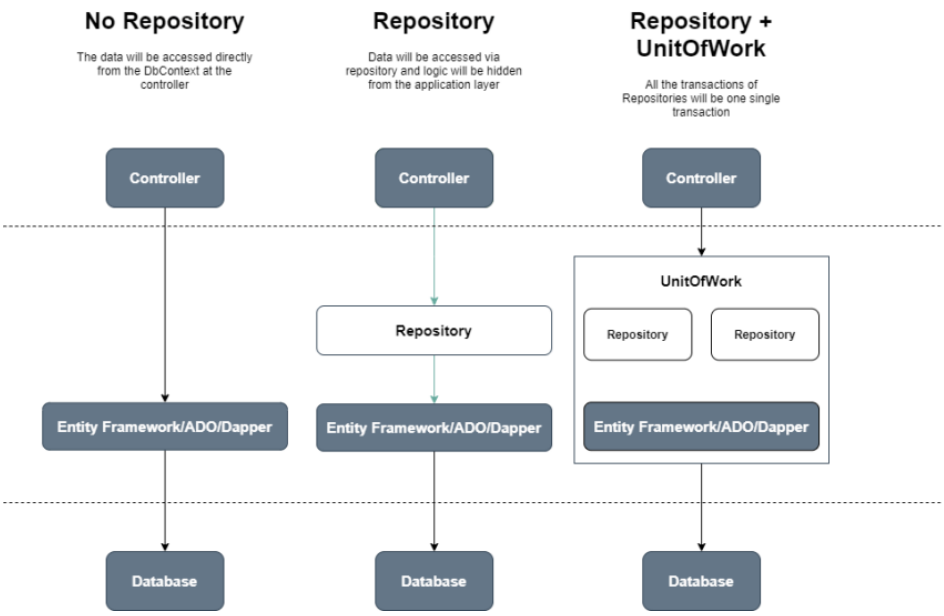
- The **Unit of Work** pattern "maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems." **UoW** orchestrates atomic data operations that span more than one repository. Think of a unit of work as the director (uow) of an orchestra (repositories).

Imagine you had a button called "Delete old records". The button would (roughly) do this:

```
using (var uow = GetUnitOfWork())
{
    var repository = uow.GetRepository<MyRecord>();
    var oldRecords = repository.Entities
        .Where(x => x.Old)
        .ToList();
    foreach(var record in oldRecords)
    {
        repository.Delete(record);
    }
    uow.Commit(); // sometimes this is optional (default behavior could be to commit)
}
```

The responsibilities of the Unit of Work are to:

- Manage transactions.
- Order the database inserts, deletes, and updates.
- Prevent duplicate updates. Inside a single usage of a Unit of Work object, different parts of the code may mark the same Invoice object as changed, but the Unit of Work class will only issue a single UPDATE command to the database.



P.S. The Entity Framework `DbContext` **is** a unit of work, and its `DbSet<T>` properties **are** repositories.

Q20: What are some *disadvantages* of Dependency Injection?

☆☆☆☆

Topics: Design Patterns Dependency Injection

Answer:

Dependency injection is, like most patterns, a **solution to problems**. So start by asking if you even have the *problem* in the first place. If not, then using the pattern most likely will make the code *worse*.

So the problems you shall experience to consider the use of DI:

1. You know you'll be doing automated testing using mock implementations
2. You know you'll be doing unit testing using mock implementations to avoid hardcoded dependencies
3. You think you might have more than one implementation in the future or the implementation might change
4. You using global services with side effects like a logger and might change it later

Also, consider:

1. Using DI does not simplify **automated testing** as much as advertised. In short, the mock implementation does NOT let you validate that the class will behave as expected with a real implementation of the interface.
2. DI makes it much harder to **navigate through code**. The problem is often finding the actual implementation being used. This can turn a simple bug fix into a day-long effort (especially for a new person on the project).

3. The use of DI and frameworks comes at the expense of needing detailed knowledge of the particular DI framework. Understanding how dependencies are loaded into the framework and adding a new dependency into the framework to inject can require reading a fair amount of background material and time spent.
4. The more you inject, the longer your startup times are going to become. Most DI frameworks create all injectable singleton instances at startup time, regardless of where they are used.
5. Dependency injection, outside of cases where it is truly needed, is also a warning sign that other superfluous code may be present in great abundance. Executives are often surprised to learn that developers add complexity for the sake of complexity.

FullStack.Cafe - Kill Your Tech Interview

Q1: How should I add an object into a collection maintained by *Aggregate Root*? ☆☆☆

Topics: Design Patterns

Problem:

Let's say I have a `BlogPost` aggregate root. it holds a `List<Comment>`. How should the BlogPost AddComment signature look?

```
public void AddComment(Comment comment)
{
    Comments.Add(comment);
}
```

or should I avoid creating references to root's children outside of it, and do something like this:

```
public void AddComment(string text, string email)
{
    Comment comment = new Comment(text, email);
    Comments.Add(comment);
}
```

Solution:

If you believe in DDD, it's perfectly fine to know about some entity beneath the aggregate root, as long as you do not store an ID or a reference to it somewhere outside of the aggregate.

I would go for the `blogPost.AddComment(new Comment(...))`-version.

Q2: Explain difference between the *Facade*, *Proxy*, *Adapter* and *Decorator* design patterns? ☆☆☆☆☆

Topics: Design Patterns

Answer:

- **Adapter** adapts a given class/object to a new interface. In the case of the former, multiple inheritance is typically employed. In the latter case, the object is wrapped by a conforming adapter object and passed around. The problem we are solving here is that of non-compatible interfaces.
- **Facade** is more like a simple gateway to a complicated set of functionality. You make a black-box for your clients to worry less i.e. make interfaces simpler.
- **Proxy** provides the same interface as the proxied-for class and typically does some housekeeping stuff on its own. (So instead of making multiple copies of a heavy object X you make copies of a lightweight proxy P which in turn manages X and translates your calls as required.) You are solving the problem of the client from having to manage a heavy and/or complex object.
- **Decorator** is used to add more gunpowder to your objects (note the term objects -- you typically decorate objects dynamically at runtime). You do not hide/impair the existing interfaces of the object but simply

extend it at runtime

Q3: What's the difference between the *Dependency Injection* and *Service Locator* patterns? ☆☆☆☆☆

Topics: Design Patterns Dependency Injection

Answer:

- With the **ServiceLocator**, the class is still responsible for creating its dependencies. It just uses the service locator to do it.
- **Service locators** hide dependencies - you can't tell by looking at an object whether it hits a database or not (for example) when it obtains connections from a locator.
- With **DI**, the class is given it's dependencies. It neither knows, nor cares where they come from.

One important result of this is that the DI example is much easier to unit test -- because you can pass it mock implementations of its dependent objects. You could combine the two -- and inject the service locator (or a factory), if you wanted.

Q4: Could you explain the difference between *Façade* vs. *Mediator*? ☆☆☆☆☆

Topics: Design Patterns

Answer:

- The **facade** only exposes the existing functionality from a different perspective.
- The **mediator** "adds" functionality because it combines different existing functionality to create a new one.

Q5: What is the difference between the *Template* patterns and the *Strategy* pattern? ☆☆☆☆☆

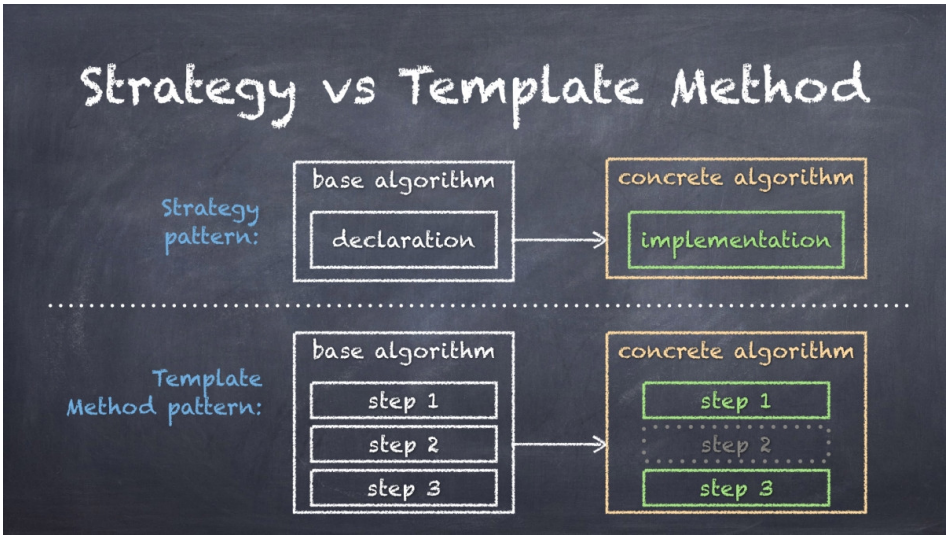
Topics: Design Patterns

Answer:

The main difference between the two is when the concrete algorithm is chosen.

- With the **Template method pattern** this happens at *compile-time* by subclassing the template. Each subclass provides a different concrete algorithm by implementing the template's abstract methods. When a client invokes methods of the template's external interface the template calls its abstract methods (its internal interface) as required to invoke the algorithm.

- In contrast, the **Strategy pattern** allows an algorithm to be chosen at *runtime* by containment. The concrete algorithms are implemented by separate classes or functions which are passed to the strategy as a parameter to its constructor or to a setter method. Which algorithm is chosen for this parameter can vary dynamically based on the program's state or inputs.



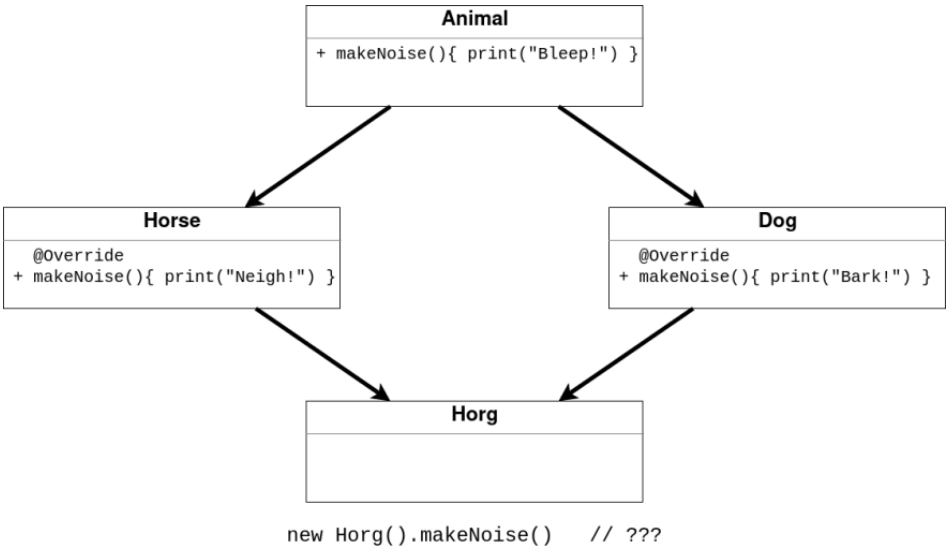
Q6: Could you explain what is the *Deadly Diamond of Death*?

☆☆☆☆☆

Topics: Design Patterns

Answer:

The **diamond problem** (sometimes referred to as the "deadly diamond of death") is an ambiguity that arises when two classes `B` and `C` inherit from `A`, and class `D` inherits from both `B` and `C`. If there is a method in `A` that `B` and `C` have overridden, and `D` does not override it, then which version of the method does `D` inherit: that of `B`, or that of `C`?



Languages have different ways of dealing with these problems of repeated inheritance. For example **Go** prevents the diamond problem at compile time:

- If a structure `D` embeds two structures `B` and `C` which both have a method `F()`, thus satisfying an interface `A`, the compiler will complain about an "ambiguous selector" if `D.F()` is called, or if an instance of `D` is assigned to a variable of type `A`.
- `B` and `C`'s methods can be called explicitly with `D.B.F()` or `D.C.F()`.

Q7: Can we use the *CQRS* without the *Event Sourcing*? ☆☆☆☆☆

Topics: DDD Software Architecture Design Patterns

Answer:

Event Sourcing (ES) is optional and in most cases complicates things more than it helps if introduced too early. Especially when transitioning from legacy architecture and even more when the team has no experience with **CQRS**.

Most of the advantages being attributed to **ES** can be obtained by storing your events in a simple Event Log. You don't have to drop your state-based persistence, (but in the long run you probably will, because at some point it will become the logical next step).

My recommendation: Simplicity is the key. Do one step at a time, especially when introducing such a dramatic paradigm shift. Start with simple CQRS, then introduce an Event Log when you (and your team) have become used to the new concepts. Then, if at all required, change your persistence to Event Sourcing.