coggle
made for free at coggle.it

Programming design patterns

Creational patterns

Provides the interface for creating families of related objects without specifying their certain classes.

When to use

When the system shouldn't be dependent on the way creation and a layout of new objects.

When objects to be created must be used together and they are related.

Participants

Factory(abstract class/interface) - Defines methods for creating objects. The methods return abstract product objects, not certain implementations.

SpecificFactories(implementations of Factory) - Implements abstract methods from the basic class and defines which certain objects to use.

Products(abstract classes/interfaces of products to be created) - Some abstract objects to be created.

SpecificProducts(implementations of Products) - Some certain object implementations to be created.

Abstract factory

Provides the interface for creating families of related objects without specifying their certain classes.
The example is in the "Abstract factory" project.
LINK

When to use

When it is not possible to know in advance what type of object is required to create.

When the system shouldn't be dependent on a process of creation of objects and should be expandable.

When required, a base class has to delegate the creation of objects to derived classes.

Factory method

Defines the interface for some object creation but the decision what type of class will be created makes a heir class. Base class delegates the creation of objects to derived classes.
The example is in the "Factory method" project.
LINK

Participants

Creator (abstract class/interface) - Defines an abstract factory method which returns a new Product object.

Product (abstract class/interface) - Defines class interface which has to be created.

ProductCreators (Implementations of Creator) - Certain product creators.

CertainProducts (implementations of Product) - Certain Product objects.

Builder

Encapsulates creation of an object and allows separate it to varied stages.
The example is in the "Builder" project.
LINK

When to use

When the creating process of an object shouldn't be dependent on what parts the object consists of and how these parts are related.

When required to get varied options of an object during its creation.

Participants

Builder (abstract class/interface) - Defines the interface to create different object's parts.

Product - Presents an object to be created.

SpecificBuilders (implementation of Builder) - Certain builder implementations.

Director - Disposer - makes an object, using builder objects.

Prototype

Allows to create objects based on earlier created objects.
The example is in the "Prototype" project.
LINK

When to use

When the certain object type to be created must be defined dynamically during the execution.

When it is not desirable to create a hierarchy of factories for creating product-objects from parallel classes hierarchy.

When it is preferable to clone and not creating and instantiating objects by constructor

Participants

SpecificPrototype - Certain object.

Prototype (abstract class/interface) - Defines the interface to clone itself. Defines the method IPrototype Clone(); which must be implemented in derived classes.

.Net provides the functionality for cloning by MemberwiseClone. However it should be counted that reference types wouldn't be copied instead of the object copy it makes the copy of the link. To resolve it the deep copy must be used.

Singleton

Ensures that only one object of a certain class will be created and gives the entry point to that object.
The example is in the "Singleton" project.
LINK

When required only one instance of a certain class is created. If the instance exists Singleton returns this existing instance.

When to use

Structural patterns

Classes and objects make more complicated classes and objects by structure and character.

Proxy

Provides an proxy-object which controles an access to another object.

When to use

When it is required to note the number of the references or provide a thread-safe operation with an object.

When it is required an access control to an object. Its name is **Protection proxy.**

When it is required to interact by network and proxy-object has to imitate another object's behaviour. It could reduce overheads. Its name is **Remote proxy.**

When it is necessary a resource access control which creation is too expensive. The real object is only created when it is really necessary. Its name is **Virtual proxy.**

Participants

AbstractSubject (abstract class/interface) - A common interface for proxies and objects.

ProxySubject (implements an **AbstractSubject**) - A proxy to access the real object.

RealSubject - A real object.

Client - Uses a proxy subject to take access to the **RealSubject**.

Adapter

Its purpose is to transform one class's interface to another class's interface.
The example is in the "Adapter" project.
LINK

When to use

When it is required to use an existing class but the interface which is used by this class doesn't match needs.

When it is required to use existing classes together but their interfaces are not compatible.

Participants

Target - Target class to which adaptation is necessary.

Adaptable - A class to be adapted.

Adapter (implements the Target) - Adapter to target.

Client - uses Target objects to implement its purposes.

Bridge

Allows to separate an abstraction from implementation. Then they could be changed independent of each other.
The example is in the "Bridge" project.
LINK

when to use

When it is required to avoid tie implementation to abstraction.

When changes in abstraction should not lead to changes in implementation.

Participants

Client - Uses abstraction objects.

CommonAbstraction (abstract class/interface) - Defines the base interface and keeps reference to the Implementor's object. Operation execution is delegated to Implementor's methods.

RefinedAbstraction (abstract class/interface) - More specific abstraction. Expands inherited interface.

Implementor (abstract class/interface) - Defines the base interface for certain implementations. Usually Implementor only defines primitive operations. More complicated operations are defined in **CommonAbstraction**.

SpecificImplementors (implement an Implementor) - Certain implements of an **Implementor**.

Composite

Unites objects groups to a tree structure as "part-whole" and allows clients to interact with separated objects or with a group of objects.
The example is in the "Composite" project.
LINK

When to use

When objects have to be grouped as a tree structure.

When clients have to be allowed to work with whole objects and with their separate parts.

Participants

Composite (implements a **Component**) - Presents a component which has an ability to contain other components and implements functions to add and remove components.

Component (abstract class/interface) - Defines each component for tree structure.

Leaf (implements a Component) - Presents a **component** which can't contain other components.

Client - Uses components.

Decorator

Allows adding new functionality to an object dynamically.
Provides a more flexible alternative to inheritance.
The example is in the "Decorator" project.
LINK

When to use

When it is required adding a new functionality dynamically.

When it is not possible to use inheritance.

Participants

Component(abstract class/interface) - Defines the interface for objects to be inherited.

SpecificComponent (implements a **Component**) - A certain **component** implementation to which a new functionality is added with a decorator.

AbstractDecorator (abstract class/interface) - Defines Decorator which inherits a component functionality. Also it keeps the reference to the base component to be decorated.

SpecificDecorators (implement the **AbstractDecorator**) - Define extra functionality for the base object.

Facade

Allows to hide the system difficulty by an easier interaction interface.
The example is in the "Facade" project.
LINK

When to use

When it is necessary to decrease the number of dependencies between client and a complicated system.

To make the work with the system easier.

When it is required to define subsystems of the components in the complicated system.

Participants

Facade - A facade which provides the interface to the client.

Subsystems - Components of the system.

Client - Uses the system.

Flyweight

Allows to use separated objects in several contexts.
The example is in the "Flyweight" project.
LINK

When to use

When an app uses a lot of similar objects.

When a changeable part of an object can be taken off to outside. This allows you to replace a plurality of objects with a small group of commonly separated objects.

Participants

Flyweight (abstract class/interface) - Defines the interface which helps flyweights objects get an outside state or affect the state.

FlyweightFactory - Creates separated **flyweights**. Because **flyweights** are separated a client doesn't have to create by itself. All created objects are saved in the objects pool. If a required **flyweight** doesn't exist the factory creates it.

Client - Uses **Flyweight** objects. Can save an external state and pass it as an argument into Flyweight methods.

SpecificFlyweight (implements a **Flyweight**) - A certain separated **Flyweight** implementation. If necessary, it adds an internal state. Also this state should be independent of a context.

UnsharedSpecificFlyweight (implements a **Flyweight**) - A certain implementation of a **Flyweight** but this kind of object is unseparated.

Behavioral patterns

Define algorithms and interaction between classes and objects.

Visitor

Allows to define operations for other classes objects without these classes changing.
The example is in the "Visitor" project.
LINK

When to use

When there are a lot of classes which implement different interfaces and it is required to make a set of operations with each of them.

When it is required to add the same set of operations to classes without these classes changing.

When there is an often operations addition to classes but the structure of these classes is stable and almost isn't changed.

When this pattern is used two classes hierarchies are defined.

For visitors who define this operation.

For elements for whom the operations are defined.

Participants

Element (abstract class/interface) - Element's interface defines an Accept method which takes a visitor as a parameter.

SpecificElements (implement an **Element**) - Certain implementations of the **Element**

ObjectStructure - Some structure which stores **Element** objects and provides access to them.

Visitor (abstract class/interface) - Visitor's interface which defines Visit method for each element.

SpecificVisitors (implement a **Visitor**) - Certain **visitors** which implement the Visitor() method.

Template Method

Defines a common subclasses acting algorithm and allows these subclasses to define particular steps of a common algorithm without algorithm structure changes.
The example is in the "Template_Method" project
LINK

When to use

If subclasses will have to define particular steps of the algorithm without structural changes of the algorithm.

If there is a duplicated code in classes which use the similar algorithm.

Participants

CommonAlgorithm (abstract class/interface) - Defines the template method which implements the algorithm. The algorithm can include the sequence of methods, some of them would be abstract/virtual and can be overridden in subclasses. The template method should be closed for overridden.

Strategy

Defines a set of algorithms, encapsulates them and ensures their permutability. The changing of an algorithm happens independent of an object which uses this algorithm.
The example is in the "Strategy" project
LINK

When to use

When there are several related classes differ in behaviour.

When required to provide a choice from several algorithms then the chosen algorithm can be changed by conditions.

When required to change object behaviour during the execution.

When a class which uses the functionality has to know nothing about the implementation of this functionality.

Participants

AbstractStrategy (abstract class/interface) - Defines an object for implementing a certain strategy.

SpecificStrategies (Implement Strategy) - Certain strategies implement algorithms

Context - Keeps the reference to an abstract **strategy** object and has an aggregate relationship with **AbstractStrategy**.

Observer

A template which uses a relationship "one-to-many". There is one observable object and plural observer objects. When the observable is changed the notification of observers happens. Also this pattern has the name "Publisher - Subscriber".
The example is in the "Observer" project.
LINK

When to use

When a common scheme of interaction assumes two sides: One of them sends messages and it is the main, another side gets messages and reacts to them.

When there is one sending object and plural subscribers. Also the count of subscribers isn't known in advance.

When objects must be in a concerted state.

Participants

Observer (abstract class/interface) - Defines an observer which subscribes to an observable object.

Observable (abstract class/interface) - Defines an observable object.

SpecificObservers (implement **Observer**) - Certain observer.

SpecificObservables (implement **Observable**) - A certain implementations of Observable.

State

Allows an object to change its acting depending on the inner state.
The example is in the "State" project.
LINK

When to use

When an object's behaviour depends on its condition and can be changed during the execution.

When there are plural condition contractions and the choice of which depends on the object's internal condition.

Participants

State(abstract class/interface) - Defines the interface of the state.

Context - an object, the behaviour of which is changed depending on the object condition. An execution of certain actions is delegated to the state object.

CertainStates (implements of the State) - certain Implementations of the **State**.

Memento

Allows to take internal object condition outside for possible further object restoring without encapsulate violation.
The example is in the "Memento" project.
LINK

When to use

If it is required to restore an object.

When an object state saving has to be without encapsulation violation.

Participants

Memento - Stores the state of the Originator object and allows the main access only to this Originator object.

Originator - Makes a memento object to save its, Originator's, state.

Caretaker - Only does save Memento object. This object doesn't have the access and operations for memento.

Iterator

Provides an abstract interface for sequenced access to all elements of a composite object without the object's inner structure disclosure.
The example is in the "Iterator" project.
LINK

When to use

If it requires to make a detour an object without the inner structure disclosure.

When there is a set of composite objects and it is required to provide the interface for detouring them.

If it is required to provide several options to detoring the same object.

Participants

Iterator (abstract class/interface) - Defines the interface to detouring objects.

Aggregate (abstract class/interface) - Defines the interface for creation of the iterator-object.

SpecificIterator - Define specific implementation of the **iterator** for detouring the aggregate object.

SpecificAggregates - Specific implementation of the **aggregator**. It keeps objects to be iterated.

Mediator

Provides objects interaction without references to each other.
The example is in the "Mediator" project.
LINK

When to use

When there are a lot of objects whose connections are difficult and confusing.

When it is required to use an object repeatedly but it is difficult because of strong ties with other objects.

Participants

AbstractMediator (abstract class/interface) - Defines the interface for objects interacting.

SpecificObjects (implements **AbstractObject**) - Certain **AbstractObjects** implementations.

AbstractObject (abstract class/interface) - Defines the interface for interacting with **mediator** object.

SpecificMediator (implements **AbstractMediator**) - A certain **AbstractMediator** implementation.

Command

Encapsulates a request to execute a certain action. Commands can get parameters, be in the queue and could be aborted.
The example is in the "Command" project.
LINK

When to use

When required to send a certain action as a parameter called as an answer to another action.

When required to insure an execution a queue of requests and their possible abort.

When needed to support the changes logging as a result of actions.

Participants

Command (abstract class/interface) - Defines the command. Usually has an "Execute" method to execute an action and often contains an "Undo" method.

Receiver - **Command** recipient. Defines an action which will be executed during a request.

SpecificCommand (implements **Command**) - Certain **command**.

Invoker - **Command** initiator. Invokes a **command** for executing a certain request.

Client - Creates a **command** and sets the receiver. Usually by the "SetCommand" method.

Interpreter

Defines presentation for another language grammar and interpreter of sentences of this language.
The example is in the "Interpreter" project.
LINK

When to use

When there are often iterated operations.

Participants

Expression(abstract class/interface) - Defines the interface of expression. Provides Interpreter() method.

TerminalExpression (implements the **Expression**) - A terminal expression which implements the Interpreter() method for terminal grammar symbols. For each grammar symbol its own **TerminalExpression** object is created.

NonterminalExpression (implements the **Expression**) - A non terminal expression, defines a grammar rule. For each grammar rule its own **NonterminalExpression** object is created.

Client - Builds sentences of the language with this grammar as an abstract syntactic tree whose nodes are terminal and nonterminal objects.

Context - Contains common information for interpreter. Terminal and non terminal objects can use context to save the operations state and the access to these states.

Chain of responsibility

All possible request handlers make the chain, the request goes through this chain. Each object in the chain decides to terminate a request processing or to send the request further to the next object.
The example is in the "Chain_of_responsibility" project.
LINK

Flaws

It doesn't mean the request will be processed in the end. If there isn't a handler which can process the request the request will just quit without processing.

When to use

When it is required to pass the request without knowing to which object.

When there is more than one object to process the request.

When the set of objects are made dynamically.

Patricipants

Client - Sends the request to the handler object.

Handler (abstract class/interface) - Defines the interface for processing a request. Also it can define a reference to the next handler.

SpecificHandlers - are certain implementations of the abstract **handler**. Contain specific actions for request processing.