NET. 022
Design Patterns

Presented By: John Hinz

@jhinz

codeinbits.com

HUMBER

WE ARE HUMBER

Agenda

- About Patterns
  - Creational
  - Behavioral
  - Structural

About Patterns

- Design patterns were introduced in Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides's seminal work Design Patterns: Elements of Reusable Object Oriented Software (Addison-Wesley).
- The book specifies and describes 23 patterns that form the foundation of any study of the subject, which are still regarded as the essential core patterns today

About Patterns (cont.)

- The 23 patterns are divided into three groups: creational, structural, and behavioral.
- **Structural patterns** are concerned with how classes and objects are composed to form larger structures.
- The **creational patterns** aim to separate a system from how its objects are created, composed, and represented.
- **Behavioral patterns** are concerned with algorithms and communication between them.

Structural Patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- There are **seven patterns** that make up the structural group, each with the role of building flexibility, longevity, and security into computer software.

Decorator (Structural)

- The role of the Decorator pattern is to provide a way of attaching new state and behavior to an object dynamically.
- The object does not know it is being “decorated,” which makes this a useful pattern for evolving systems. A key implementation point in the Decorator pattern is that decorators both inherit the original class and contain an instantiation of it.
- https://dotnetfiddle.net/oCZjmC

Proxy (Structural)

- The Proxy pattern supports objects that control the creation of and access to other objects. The proxy is often a small (public) object that stands in for a more complex (private) object that is activated once certain circumstances are clear.
- https://dotnetfiddle.net/YcG7hY

Bridge (Structural)

- The Bridge pattern decouples an abstraction from its implementation, enabling them to vary independently.
- The Bridge pattern is useful when a new version of software is brought out that will replace an existing version, but the older version must still run for its existing client base.
- https://dotnetfiddle.net/REFPxF

Composite (Structural)

- The Composite pattern arranges structured hierarchies so that single components and groups of components can be treated in the same way. Typical operations on the components include add, remove, display, find, and group.
- https://dotnetfiddle.net/93GSEy

Flyweight (Structural)

- The Flyweight pattern promotes an efficient way to share common information present in small objects that occur in a system in large numbers.
- The Flyweight pattern distinguishes between the *intrinsic* and *extrinsic* state of an object.

Flyweight (Structural) (cont.)

- The greatest savings in the Flyweight pattern occur when objects use both kinds of state but:
  - The intrinsic state can be shared on a wide scale, minimizing storage requirements.
  - The extrinsic state can be computed on the fly, trading computation for storage.
- https://dotnetfiddle.net/FiBWLe

Adapter (Structural)

- The Adapter pattern enables a system to use classes whose interfaces don’t quite match its requirements.
- Many examples of the Adapter pattern involve input/output because that is one domain that is constantly changing.
- https://dotnetfiddle.net/gMCsog

Façade (Structural)

- The role of the Façade pattern is to provide different high-level views of subsystems whose details are hidden from users.
- The operations that might be desirable from a user's perspective could be made up of different selections of parts of the subsystems.
- https://dotnetfiddle.net/oYUKEX

Creational Patterns

- The creational patterns aim to separate a system from how its objects are created, composed, and represented.
- They increase the system's flexibility in terms of the what, who, how, and when of object creation.
- Creational patterns encapsulate the knowledge about which classes a system uses, but they hide the details of how the instances of these classes are created and put together.

Prototype (Creational)

- The Prototype pattern creates new objects by cloning one of a few stored prototypes.
- The Prototype pattern has two advantages:
  - it speeds up the instantiation of very large, dynamically loaded classes (when copying objects is faster),
  - it keeps a record of identifiable parts of a large data structure that can be copied without knowing the subclass from which they were created.
- https://dotnetfiddle.net/I5WlyE

Factory (Creational)

- The Factory Method pattern is a way of creating objects, but letting subclasses decide exactly which class to instantiate.
- The Factory Method instantiates the appropriate subclass based on information supplied by the client or extracted from the current state.
- https://dotnetfiddle.net/Dj0V0a

Singleton (Creational)

- The purpose of the Singleton pattern is to ensure that there is only one instance of a class, and that there is a global access point to that object.
- The pattern ensures that the class is instantiated only once and that all requests are directed to that one and only object. Moreover, the object should not be created until it is actually needed.
- Singleton class itself that is responsible for ensuring this constraint, not the clients of the class.
- https://dotnetfiddle.net/q2wGsl

Abstract Factory (Creational)

- This pattern supports the creation of products that exist in families and are designed to be produced together. The abstract factory can be refined to concrete factories, each of which can create different products of different types and in different combinations.
- The pattern isolates the product definitions and their class names from the client so that the only way to get one of them is through a factory. For this reason, product families can easily be interchanged or updated without upsetting the structure of the client.
- https://dotnetfiddle.net/m4rEsE

Builder (Creational)

- The Builder pattern separates the specification of a complex object from its actual construction.
- The same construction process can create different representations.
- https://dotnetfiddle.net/j1AS8n

Behavioural Patterns

- Behavioral patterns are concerned with algorithms and communication between them.
- The operations that make up a single algorithm might be split up between different classes, making a complex arrangement that is difficult to manage and maintain.
- The behavioral patterns capture ways of expressing the division of operations between classes and optimize how the communication should be handled.

Strategy (Behavioural)

- The Strategy pattern involves removing an algorithm from its host class and putting it in a separate class. There may be different algorithms (strategies) that are applicable for a given problem. If the algorithms are all kept in the host, messy code with lots of conditional statements will result.
- The Strategy pattern enables a client to choose which algorithm to use from a family of algorithms and gives it a simple way to access it. The algorithms can also be expressed independently of the data they are using.
- https://dotnetfiddle.net/lNSvzq

State (Behavioural)

- The State pattern, can be seen as a dynamic version of the Strategy pattern.
- When the state inside an object changes, it can change its behavior by switching to a set of different operations. This is achieved by an object variable changing its subclass, within a hierarchy.
- https://dotnetfiddle.net/Qsuf6S

Template Method (Behavioural)

- The Template Method pattern enables algorithms to defer certain steps to subclasses.
- The structure of the algorithm does not change, but small well-defined parts of its operation are handled elsewhere.
- https://dotnetfiddle.net/1DxQnk

Chain of Responsibility (Behavioural)

- The Chain of Responsibility pattern works with a list of Handler objects that have limitations on the nature of the requests they can deal with.
- If an object cannot handle a request, it passes it on to the next object in the chain. At the end of the chain, there can be either default or exceptional behavior.
- https://dotnetfiddle.net/mBr3eN

Command (Behavioural)

- The Command pattern creates distance between the client that requests an operation and the object that can perform it. This pattern is particularly versatile. It can support:
  - Sending requests to different receivers
  - Queuing, logging, and rejecting requests
  - Composing higher-level transactions from primitive operations
  - Redo and Undo functionality
- https://dotnetfiddle.net/RTgPuS

Iterator (Behavioural)

- The Iterator pattern provides a way of accessing elements of a collection sequentially, without knowing how the collection is structured.
- As an extension, the pattern allows for filtering elements in a variety of ways as they are generated.
- https://dotnetfiddle.net/ggKpIP

Mediator (Behavioural)

- The Mediator pattern is there to enable objects to communicate without knowing each other's identities.
- It also encapsulates a protocol that objects can follow.
- https://dotnetfiddle.net/6iUexD

Observer (Behavioural)

- The Observer pattern defines a relationship between objects so that when one changes its state, all the others are notified accordingly.
- There is usually an identifiable single publisher of new state, and many subscribers who wish to receive it.
- https://dotnetfiddle.net/xDtkiU

Visitor (Behavioural)

- The Visitor pattern defines and performs new operations on all the elements of an existing structure, without altering its classes.
- https://dotnetfiddle.net/8REEiy

Interpreter (Behavioural)

- The Interpreter pattern supports the interpretation of instructions written in a language or notation defined for a specific purpose.
- The notation is precise and can be defined in terms of a grammar.
- https://dotnetfiddle.net/3GAWD6

Memento (Behavioural)

- This pattern is used to capture an object's internal state and save it externally so that it can be restored later.
- https://dotnetfiddle.net/M2Aovf

THANK YOU.

HUMBER

WE ARE HUMBER