Design Patterns

- A design pattern is a reusable solution to a commonly occurring problem in software design. It provides a structured approach to solving specific design challenges while promoting code reusability, maintainability, and flexibility. Design patterns capture best practices and proven techniques that help developers design robust and efficient software systems. There are several types of design patterns, each serving a specific purpose. Here are some common categories of design patterns:
    - Creational Patterns: These patterns focus on object creation mechanisms, providing flexibility in how objects are instantiated. Creational patterns are a category of design patterns that focus on object creation mechanisms. They provide flexible and reusable solutions for creating objects, hiding the details of instantiation and promoting code maintainability and extensibility. Creational patterns address the problem of creating objects in a way that enhances flexibility, decoupling, and encapsulation. Creational patterns emerged as a response to common challenges faced during object creation and initialization in software systems. Without proper design patterns, object creation can become complex, tightly coupled, and difficult to manage. Creational patterns formalize best practices and provide standardized approaches to create objects, making the process more manageable and adaptable. Examples include:
        - Singleton Pattern: Ensures only one instance of a class exists throughout the application. It provides global access to this single instance and prevents multiple instances from being created. This pattern is useful in situations where sharing a single instance of a class is important, such as managing access to a shared resource or controlling system-wide configurations.
            - Singleton class
            - Has property that holds instance of instance of singleton class
            - Private constructor initializes the class
            - Public method initializes class if not already initialized, else return singleton class
            - ```
              package main

              import "fmt"

              type Singleton struct {
               singleton *Singleton
              }

              func newSingleton() *Singleton {
               return &Singleton{
                singleton: &Singleton{},
               }
              }

              func (s *Singleton) GetSingleton() *Singleton {
              ```
              ```

```
 if s.singleton == nil {
  return newSingleton()
 }
 return s.singleton
}

func main() {
 s := Singleton{}
 fmt.Println(s.GetSingleton())
}
```

- Factory Pattern: Provides an interface for creating objects without specifying their concrete classes. The Factory pattern provides an interface or base class for creating objects, without specifying their concrete classes upfront. It allows the client code to abstract the process of object creation and delegate the responsibility to a factory class. The factory class determines the appropriate concrete class to instantiate based on the specific parameters or conditions. This pattern helps decouple the client code from the details of object creation, adding flexibility and promoting loose coupling.
  - All product options implement a base class/interface
  - FactoryClass decides type of object to create based on input
  - ```
    package main

    import "fmt"

    type Factory interface {
     displayInfo()
    }

    type Product1 struct {
     Weight string
    }

    func newProduct1(weight string) Factory {
     return Product1{
      Weight: weight,
     }
    }

    func (p Product1) displayInfo() {
     fmt.Println(p.Weight)
    }

    type Product2 struct {
     Volume string
    }

    func newProduct2(volume string) Factory {
     return Product2{
    ```
    ```

```
      Volume: volume,
     }
    }

    func (p Product2) displayInfo() {
     fmt.Println(p.Volume)
    }

    type ProductFactory struct{}

    func (f *ProductFactory) createProducts(productName
string) Factory {
     switch productName {
     case "1":
      return newProduct1(productName)
     case "2":
      return newProduct2(productName)
     default:
      fmt.Println("Wrong product. Please select 1 or 2!")
      return nil
     }
    }

    func main() {
     f := ProductFactory{}

     p1 := f.createProducts("1")
     p2 := f.createProducts("2")
       wp := f.createProducts("3")
     p1.displayInfo()
     p2.displayInfo()
       fmt.Println(wp)
    }
```

- Builder Pattern: Separates the construction of complex objects from their representation. The Builder pattern separates the construction of complex objects from their representation. It provides a way to create objects step by step, allowing different variations to be constructed using the same building process. The central idea is to have a builder object that encapsulates the construction logic and exposes methods to set different attributes or properties of the object being built. The Builder pattern is useful when constructing complex objects where the construction process involves multiple steps or different configurations.
    - Product class is initialized with required attributes
    - Constructor to initialize product with default attributes, and Methods to set optional attributes for product are provided
    - ProductBuilder returns default product, and allows product with optional attributes to be built

- ```
  package main

  import "fmt"

  type Product struct {
   Name        string
   Description string
   Price       int64
   Stock       int
  }

  func newProduct(name string) Product {
   return Product{
    Name: name,
   }
  }

  func (p *Product) setDescription(description string) {
   p.Description = description
  }

  func (p *Product) setPrice(price int64) {
   p.Price = price
  }

  func (p *Product) setStock(stock int) {
   p.Stock = stock
  }

  func (p *Product) displayInfo() {
   fmt.Printf("%+v\n", p)
  }

  type ProductBuilder struct {
   product Product
  }

  func NewProductBuilder(name string) *ProductBuilder {
   return &ProductBuilder{
    product: Product{
     Name: name,
    },
   }
  }

  func (p *ProductBuilder) withDescription(description
  string) *ProductBuilder {
   p.product.Description = description
   return p
  }

  func (p *ProductBuilder) withPrice(price int64)
  *ProductBuilder {
  ```
  ```

```
    p.product.Price = price
    return p
   }

   func (p *ProductBuilder) withStock(stock int)
*ProductBuilder {
    p.product.Stock = stock
    return p
   }

   func (p *ProductBuilder) build() Product {
    return p.product
   }

   func main() {
    prod1 := NewProductBuilder("Cheese").build()
    prod2 :=
NewProductBuilder("Biscuit").withDescription("Tasty").withPri
ce(10).withStock(0).build()

    prod1.displayInfo()
    prod2.displayInfo()
   }
```

- [[TODO]] Factory Method Pattern
- [[TODO]] Abstract Factory Pattern
- [[TODO]] Prototype Pattern
- [[TODO]] Object Pool
- [[TODO]] Curiously Recurring Template Pattern
- [[TODO]] Dependency Injection

- Structural Patterns: These patterns deal with object composition, helping to form larger structures and provide flexible relationships between objects. Examples include:
  - Adapter Pattern: Adapter Pattern is used when to make two incompatible interfaces work together. It allows objects with different interfaces to collaborate and interact seamlessly. Imagine an existing class (or system) that cannot be modified, and it should work with a new class or interface. An "adapter" class can be created, that acts as an intermediary. The adapter class implements the interface expected by the client code and delegates the calls to the adapted class. It "adapts" the interface of the existing class to the one expected by the client.
    - Legacy class take in dimensions in inches as string, returns area in sq inches as string
    - NewInterface takes dimensions in centimeters

- AdapterClass implements NewInterface, constructor initializes Legacy class, converts centimeters to inches, returns area in sq cms

- ```
  package main

  import (
   "fmt"
   "strconv"
  )

  const (
   inchesInCentimeters     = 2.54
   sqCentimetersInSqInches = 6.4516
  )

  type LegacySystem struct {
   Length, Breadth, Height string
  }

  func (l *LegacySystem) volume() string {
   len, _ := strconv.ParseFloat(l.Length, 32)
   b, _ := strconv.ParseFloat(l.Breadth, 32)
   h, _ := strconv.ParseFloat(l.Height, 32)
   return fmt.Sprint(len * b * h)
  }

  type NewSystem interface {
   volume(length, breadth, height float32) float32
  }

  type Adapter struct {
   legacySystem LegacySystem
  }

  func newAdapter(legacySystem LegacySystem) *Adapter {
   return &Adapter{
    legacySystem: legacySystem,
   }
  }

  func (a *Adapter) volume(length, breadth, height float32)
  float32 {
   var l, b, h float32
   l, b, h = length/inchesInCentimeters,
  breadth/inchesInCentimeters, height/inchesInCentimeters
   a.legacySystem.Length = fmt.Sprint(l)
   a.legacySystem.Breadth = fmt.Sprint(b)
   a.legacySystem.Height = fmt.Sprint(h)
   volume, _ := strconv.ParseFloat(a.legacySystem.volume(),
  32)
   return float32(volume) * sqCentimetersInSqInches
  }
  ```
  ```

```
func main() {
 legacySystem := LegacySystem{}
 adapter := newAdapter(legacySystem)
 fmt.Println(adapter.volume(10, 15, 20))
}
```

- Decorator Pattern: Decorator Pattern is used to dynamically add or modify the behavior of objects without altering their class structure. It's a way to extend the functionality of objects at runtime. This pattern is used when adding new responsibilities or features to a base object or component without altering its source code. Decorator pattern involves a set of decorator classes that are used to wrap concrete components. These decorators add additional behaviors to the wrapped component while keeping the component's interface intact.
  - Have base interface implemented by base class
  - Decorator also implementes base interface, can take base class as input to call methods on base class
  - Additional decorators extend Decorator, can call super.operation
  - ```
    package main

    import "fmt"

    type TextFormatter interface {
     Edit(text, textType string)
    }

    type Editor struct {
     Text, Type string
    }

    func (e *Editor) Edit(text, textType string) {
     e.Text = text
     e.Type = textType
     fmt.Printf("Text: %s, Type: %s\n", e.Text, e.Type)
    }

    type TextFormatters struct {
     Component TextFormatter
    }

    func (t *TextFormatters) Edit(text, textType string) {
     t.Component.Edit(text, textType)
    }

    type Italics struct {
     Component TextFormatter
    ```
    ```

```
}

func (i *Italics) Edit(text, textType string) {
 i.Component.Edit(text, "italics")
}

type Bold struct {
 Component TextFormatter
}

func (b *Bold) Edit(text, textType string) {
 b.Component.Edit(text, "bold")
}

type Underlined struct {
 Component TextFormatter
}

func (u *Underlined) Edit(text, textType string) {
 u.Component.Edit(text, "underlined")
}

func main() {
 baseEditor := &Editor{}
 baseEditor.Edit("This is a text.", "normal")

 decoratedEditor := &Italics{
  Component: &TextFormatters{
   Component: baseEditor,
  },
 }

 decoratedEditor.Edit("Formatted text", "")

 decoratedEditor = &Italics{
  Component: &Bold{
   Component: &TextFormatters{
    Component: baseEditor,
   },
  },
 }

 decoratedEditor.Edit("Formatted text", "")

 decoratedEditor = &Italics{
  Component: &Underlined{
   Component: &TextFormatters{
    Component: baseEditor,
   },
  },
 }

 decoratedEditor.Edit("Formatted text", "")
}
```

- Composite Pattern: - The Composite Pattern is used to treat individual objects and compositions of objects (composites) uniformly. It allows building of complex structures from simple objects while allowing clients to work with both individual objects and compositions seamlessly. In this pattern, there's a common interface for both leaf nodes (individual objects) and composite nodes (compositions of objects). Each component, whether a leaf or a composite, implements this interface. Clients can interact with these components without knowing whether they are dealing with a single object or a complex composition. This pattern is particularly useful when you need to represent hierarchical structures.
    - Base interface for leaf nodes
    - Implemented by leaf class
    - Composite class has list of leaves
    - For each child in list, call methods
    - ```
      package main

      import "fmt"

      type Component interface {
       operation(name, leafType string)
      }

      type Leaf struct {
       Name, Type string
      }

      func (l *Leaf) operation() {
       fmt.Println("Name: ", l.Name, "Type: ", l.Type)
      }

      type Composite struct {
       Leaves map[string]Leaf
      }

      func (c *Composite) operation() {
       for _, value := range c.Leaves {
        value.operation()
       }
      }

      func (c *Composite) add(leaf Leaf) {
       c.Leaves[leaf.Name] = leaf
      }

      func (c *Composite) remove(leaf Leaf) {
       delete(c.Leaves, leaf.Name)
      ```
      ```

```
}

func main() {
 leaf1 := Leaf{
  Name: "One",
  Type: "1",
 }
 leaf2 := Leaf{
  Name: "Two",
  Type: "2",
 }
 composite := Composite{}
 composite.Leaves = make(map[string]Leaf)
 composite.add(leaf1)
 composite.add(leaf2)
 composite.operation()
}
```

        - [[TODO]] Adapter Pattern
        - [[TODO]] Facade Pattern
        - [[TODO]] Flyweight Pattern
        - [[TODO]] Bridge Pattern
    - Behavioral Patterns: These patterns focus on communication between objects and the assignment of responsibilities among them. Examples include:
        - Observer Pattern: The Observer Pattern defines a one-to-many dependency between objects, where one object (the subject) maintains a list of its dependents (observers) and notifies them of any state changes.
            - Subject interface with attach, detach and notify methods
            - Subject class has observers list
            - Observer interface has update method
            - Observer class has injected Subject
            -

```
package main

import "fmt"

type Subject interface {
 attach(Observer)
 detach(Observer)
 notify(val interface{})
}

type ConcreteSubject struct {
```

```
    observers map[string]ConcreteObserver
}

func (c *ConcreteSubject) attach(obs ConcreteObserver) {
    c.observers[obs.name] = obs
}

func (c *ConcreteSubject) detach(obs ConcreteObserver) {
    if _, ok := c.observers[obs.name]; ok {
        delete(c.observers, obs.name)
    }
}

func (c *ConcreteSubject) notify(val interface{}) {
    for key, observer := range c.observers {
        observer.update(fmt.Sprintf("generating new notification
for observer %v at key %v: %v", observer.name, key, val))
    }
}

type Observer interface {
    update(val interface{})
}

type ConcreteObserver struct {
    name    string
    subject Subject
}

func (c *ConcreteObserver) update(val interface{}) {
    fmt.Println(val)
}

func main() {
    pub := &ConcreteSubject{
        observers: make(map[string]ConcreteObserver),
    }
    obs1, obs2 := ConcreteObserver{}, ConcreteObserver{}
    obs1.name = "first"
    obs2.name = "second"
    pub.attach(obs1)
    pub.attach(obs2)
    fmt.Println(pub.observers)

    for i := 0; i < 10; i++ {
        fmt.Println("updated value: ", i)
        pub.notify(i)
    }
}
```

- Strategy Pattern: The Strategy Pattern allows you to define a family of algorithms, encapsulate each one, and make them interchangeable. It enables the client to choose the

appropriate algorithm at runtime.

- Strategy interface has doAlgorithm() method
- Separate concrete strategy classes that provide different implementations of doAlgorithm()
- Context class accepts strategy and sets the strategy context, and calls doAlgorithm() in that context
- ```
  package main

  import "fmt"

  type Strategy interface {
   doAlgorithm()
  }

  type BinarySort struct{}

  func (b *BinarySort) doAlgorithm() {
   fmt.Println("sorted list of numbers using binary sort")
  }

  type MergeSort struct{}

  func (m *MergeSort) doAlgorithm() { fmt.Println("sorted
  list of numbers using merge sort") }

  type Context struct {
   strategy Strategy
  }

  func (c *Context) setStrategy() {}

  func (c *Context) executeStrategy() {
   c.strategy.doAlgorithm()
  }

  func main() {
   ctx := Context{
    strategy: &BinarySort{},
   }
   ctx.setStrategy()
   ctx.executeStrategy()

   ctx = Context{
    strategy: &MergeSort{},
   }
   ctx.setStrategy()
   ctx.executeStrategy()
  }
  ```
  ```

- Command Pattern: The Command Pattern encapsulates a request as an object, allowing you to parameterize clients with different requests, queue requests, and support undoable operations. It separates the sender and receiver of a request and decouples the invoker from the receiver.
    - Command interface has execute method
    - Separate concrete classes implement Command interface, concrete classes have Receivers as properties
    - Receiver class has methods that performs separate actions for separate concrete class of Command interface
    - Invoker class has setCommand and executeCommand methods and Command as property
    - ```
      package main

      import "fmt"

      type Command interface {
       execute()
      }

      type LinearSearch struct {
       receiver Receiver
      }

      func (l *LinearSearch) execute() {
       fmt.Println("searched target using linear search")
       l.receiver.actionLinearSearch()
      }

      type BinarySearch struct {
       receiver Receiver
      }

      func (b *BinarySearch) execute() {
       fmt.Println("searched target using binary search")
       b.receiver.actionBinarySearch()
      }

      type Receiver struct{}

      func (r *Receiver) actionLinearSearch() {
       fmt.Println("Performing linear search action")
      }

      func (r *Receiver) actionBinarySearch() {
       fmt.Println("Performing binary search action")
      }
      ```
      ```

```
type Invoker struct {
 command Command
}

func (i *Invoker) setCommand() {}

func (i *Invoker) executeCommand() {
 i.command.execute()
}

func main() {
 inv := Invoker{
  command: &LinearSearch{},
 }
 inv.executeCommand()
 inv = Invoker{
  command: &BinarySearch{},
 }
 inv.executeCommand()
}
```

- [[TODO]] Mediator Pattern
- [[TODO]] Iterator Pattern
- [[TODO]] Concurrency Pattern

- Architectural Patterns: These patterns are larger in scale and address the overall structure and organization of an application or system. Examples include:
  - Model-View-Controller (MVC) Pattern: Separates the application logic into Model, View, and Controller components.
    - Model: Represents the application's data and business logic.
    - View: Responsible for presenting the data to the user.
    - Controller: Accepts user input, processes it, and communicates with the Model and View.
    -

```
class Model:
    data

    method get_data():
        return data

    method set_data(new_data):
        data = new_data

class View:
    method display_data(data):
```

```
            # Display data to the user

    class Controller:
        model
        view

        constructor(model, view):
            this.model = model
            this.view = view

        method update_view():
            data = model.get_data()
            view.display_data(data)

    # Usage
    model = Model()
    view = View()
    controller = Controller(model, view)

    # User interacts with the controller
    new_data = "Updated Data"
    controller.model.set_data(new_data)
    controller.update_view()
```

- Layered Architecture Pattern: The Layered Architecture pattern divides an application into distinct layers, each responsible for specific functionality. Common layers include presentation, business logic, and data access. It promotes separation of concerns and maintainability.

    ```
                    # Presentation Layer
                    class PresentationLayer:
                        method display_data(data):
                            # Display data to the user

                    # Business Logic Layer
                    class BusinessLogicLayer:
                        method process_data(data):
                            # Perform business logic operations
                            return processed_data

                    # Data Access Layer
                    class DataAccessLayer:
                        method retrieve_data():
                            # Retrieve data from a database or
    external source
                            return data

                    # Usage
                    presentation_layer = PresentationLayer()
                    business_logic_layer = BusinessLogicLayer()
                    data_access_layer = DataAccessLayer()
    ```

```
                data = data_access_layer.retrieve_data()
                processed_data =
business_logic_layer.process_data(data)

presentation_layer.display_data(processed_data)
                ```
```

- Microservices Pattern: The Microservices pattern structures an application as a collection of small, independent, and loosely coupled services, each responsible for a specific functionality. Microservices communicate over a network and can be developed, deployed, and scaled independently.

    ```
    # Service 1
    class Service1:
        method handle_request():
            # Process the request
            return "Response from Service 1"

    # Service 2
    class Service2:
        method make_request_to_service1():
            # Make an HTTP request to Service 1
            response = HTTP.get("http://service1/api")
            return response

        method handle_request():
            response_from_service1 =
make_request_to_service1()
            # Process the response
            return f"Response from Service 2 with
{response_from_service1}"

    # Usage
    service1 = Service1()
    service2 = Service2()

    # Simulate a client request to Service 2
    response = service2.handle_request()
    ```
    ```

- [[TODO]] Factory Method Pattern (can be architectural if used to define the architecture)
- [[TODO]] Abstract Factory Pattern (can be architectural if used to define the architecture)