Design Patterns

A **design pattern** is a general description of the solution to a well-established problem using an arrangement of classes and objects.

Patterns describe the shape of code rather than the details. They’re a means of communicating design ideas. They are not specific to any one programming language.

You’ll learn about lots of patterns in CSC301 (Introduction to Software Engineering) and CSC302 (Engineering Large Software Systems).
Gang of Four

First codified by the Gang of Four in 1995
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides

Original Gang of Four book described 23 patterns
- More have been added
- Other authors have written books
Book provides an overview of:

- **Name**
- **Problem**: when to use the pattern
  - motivation: sample application scenario
  - applicability: guidelines for when your code needs this pattern
- **Solution**:
  - structure: UML Class Diagram of generic solution
  - participants: description of the basic classes involved in generic solution
  - collaborations: describes the relationships and collaborations among the generic solution participants
  - sample code
- Consequences, Known Uses, Related Patterns, Anti-patterns
Iterator Design Pattern

Context

• A container/collection object.

Problem

• Want a way to iterate over the elements of the container.

• Want to have multiple, independent iterators over the elements of the container.

• Do not want to expose the underlying representation: should not reveal how the elements are stored.
Iterator Design Pattern: Java

```
«interface»
Iterator<T>
+ hasNext(): boolean
+ next(): T

YourIteratorClass

«interface»
Iterable<T>
+ iterator(): Iterator<T>

YourIterableClass
+ iterator(): Iterator<T>

Returns an instance of YourIteratorClass.
```
Observer Design Pattern

Problem:

• Need to maintain consistency between related objects.
• Two aspects, one dependent on the other.
• An object should be able to notify other objects without making assumptions about who these objects are.
Observer: Standard Solution

**Subject**
- observers : Observer []
  + attach(o: Observer)
  + detach(o: Observer)
  + notify()

**ConcreteSubject**
- subjectState
  + getState()
  + setState()

**Observer**
+ update()

**ConcreteObserver**
- observerState
- subject : ConcreteSubject
  + update()

for all o: observers:  
o.update()  
observerState = subject.getState()
Observer: Java Implementation

```
Observable
- observers
+ addObserver(o: Observer): void
+ deleteObserver(o: Observer): void
+ deleteObservers(): void
+ hasChanged(): boolean
+ setChanged(): void
+ clearChanged(): void
+ notifyObservers(): void
+ notifyObservers(arg: Object): void
```

```
Observer
+ update(o: Observable, arg: Object)
```

```
YourObservableClass
- state
+ getState()
+ setState()
```

```
YourObserverClass
- observerState
```
Observer: Example in Java

```java
class Observable {
    private List<Observer> observers = new ArrayList<>;

    public void addObserver(Observer o) {
        observers.add(o);
    }

    public void deleteObserver(Observer o) {
        observers.remove(o);
    }

    public void deleteObservers() {
        observers.clear();
    }

    public boolean hasChanged() {
        return observers.size() > 0;
    }

    public void setChanged() {
        hasChanged = true;
    }

    public void clearChanged() {
        hasChanged = false;
    }

    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(this, null);
        }
    }

    public void notifyObservers(Object arg) {
        for (Observer o : observers) {
            o.update(this, arg);
        }
    }
}
```

```java
interface Observer {
    void update(Observable o, Object arg);
}
```

```java
class Company {
    private String name;

    public Company(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public String toString() {
        return name;
    }
}
```

```java
class Customer {
    private String name;

    public Customer(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public String toString() {
        return name;
    }
}
```

```java
class Parcel {
    private String trackingNumber;
    private String location;

    public Parcel(String trackingNumber, String location) {
        this.trackingNumber = trackingNumber;
        this.location = location;
    }

    public String getTrackingNumber() {
        return trackingNumber;
    }

    public String getLocation() {
        return location;
    }

    public void updateLocation(String loc) {
        location = loc;
    }

    public String toString() {
        return location;
    }
}
```
Strategy Design Pattern

Problem:

- multiple classes that differ only in their behaviour (for example, use different versions of an algorithm)
- but the various algorithms should not be implemented within the class
- want the implementation of the class to be independent of a particular implementation of an algorithm
- the algorithms could be used by other classes, in a different context
- want to **decouple** — separate — the implementation of the class from the implementations of the algorithms
Strategy: Standard Solution

- **Context**
  - `strategy: Strategy`
  - `contextInterface()`

- **Strategy**
  - `algorithmInterface()`

- **ConcreteStrategyA**
  - `algorithmInterface()`

- **ConcreteStrategyB**
  - `algorithmInterface()`
Example: without the Strategy pattern
Example: using the Strategy pattern

```
Author
- name: String
- books: List<Book>
- sorter: Sorter<Book>
+ Author(name: String, sorter: Sorter) 
+ getName(): String 
+ setName(): void
+ addBook(book: Book): void
+ sortBooks(): void
+ toString(): String

Comparable<T>
compareTo(other: T): int

Book
- name: String
- isbn: String
+ Book(name: String, isbn: String)
+ getName(): String
+ setName(): void
+ getISBN(isbn: String): void
+ ISBN(): void
+ compareTo(other: Book): int
+ toString(): String

SelectionSorter<T>
+ sort(list: List<T>): void

InsertionSorter<T>
+ sort(list: List<T>): void
```