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
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: Java Implementation

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

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

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

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

    public boolean hasChanged() {
        // Check for changes
    }

    public void setChanged() {
        // Set changed
    }

    public void clearChanged() {
        // Clear changes
    }

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

    public void notifyObservers(Object arg) {
        for (Observer o : observers) {
            o.update(this, arg); // Notify observers with argument
        }
    }
}

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

class YourObservableClass {
    private String state;

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}

class YourObserverClass implements Observer {
    private String observerState;

    public void update(Observable o, Object arg) {
        // Handle update
    }
}
```
Observer: Example in Java

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

if hasChanged():
  for all o: observers:
    o.update(this, agr)
  clearChanged()

Company
- name: String
  + toString(): String

Customer
- name: String
  + toString(): String

Parcel
- trackingNumber: String
- location: String

+ getLocation(): String
+ updateLocation(loc: String): void
+ toString(): String

Observer
«interface»
+ update(o: Observable, arg: Object)
```
Uses of Observer in Java

In reality, people usually implement their own

- Usually can’t or don’t want to sub-class from Observable
- Can’t have your own class hierarchy and multiple inheritance is not available
- Has been replaced by the Java Delegation Event Model (DEM)
  - Passes event objects instead of update/notify

Listener is specific to GUI classes
Singleton Design Pattern

Context

• Classes for which only one instance should exist (singleton).
• Provide a global point of access.

Problem

• How do you ensure that it is never possible to create more than one instance of a singleton class?

Forces

• The use of a public constructor cannot guarantee that no more than one instance will be created.
• The singleton instance must be accessible to all classes that require it.
Singleton: Solution

C clients access a Singleton instance solely though Singleton's getInstance() operation.
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 (i.e., should not reveal how the elements are stored).
Iterator Design Pattern: Solution

- Iterator
  - + first()
  - + next()
  - + isDone()
  - + currentItem()

- Container
  - + createIterator(): Iterator

- YourContainerClass
  - + createIterator(): Iterator

Returns instance of YourIteratorClass.
Iterator Design Pattern: Java

Diagram showing the relationship between the `Iterator` and `Iterable` interfaces, and an example class `YourIterableClass` implementing the `Iterable` interface.
Iterator: Example in Java

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

AddressBookIterator
```

```
<<interface>>
Iterable<T>
+ iterator(): Iterator<T>

AddressBook
+ iterator(): Iterator<Contact>

Returns instance of AddressBookIterator.
```
Strategy Design Pattern

Problem:

- multiple classes that differ only in their behaviour (e.g., 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
Example: without the Strategy pattern
Example: using the Strategy pattern