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: Standard Solution

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

for all o: observers:
  o.update()

ConcreteSubject
- subjectState
  + getState()
  + setState()

Observer
+ update()

ConcreteObserver
- observerState
- subject : ConcreteSubject
  + update()

observerState = subject.getGameState()
Observer: Java Implementation

```java
public interface Observer {
    boolean hasChanged();
    void addObserver(Observer o);
    void deleteObserver(Observer o);
    void deleteObservers();
    boolean hasObservers();
    void setChanged();
    void clearChanged();
    void notifyObservers();
    void notifyObservers(Object arg);
}
```

```java
public class YourObservableClass {
    private List<Observer> observers = new ArrayList<>();

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

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

    public List<Observer> getObservers() {
        return observers;
    }

    public void setState(int state) {
        // Update state
    }

    public int getState() {
        return state;
    }
}
```

```java
public class YourObserverClass implements Observer {
    private boolean changed = false;

    @Override
    public boolean hasChanged() {
        return changed;
    }

    @Override
    public void addObserver(Observer o) {
        super.addObserver(o);
    }

    @Override
    public void deleteObserver(Observer o) {
        super.deleteObserver(o);
    }

    @Override
    public void deleteObservers() {
        super.deleteObservers();
    }

    @Override
    public void setChanged() {
        super.setChanged();
        changed = true;
    }

    @Override
    public void clearChanged() {
        super.clearChanged();
        changed = false;
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.setChanged();
        }
    }

    @Override
    public void notifyObservers(Object arg) {
        notifyObservers();
    }
}
```
Observer: Example in Java

```java
if (hasChanged()) {
    for (Observer o : observers) {
        o.update(this, agr);
    }
    clearChanged();
}
```
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

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

```
«interface»
Iterator
+ first()
+ next()
+ isDone()
+ currentItem()

YourIteratorClass

«interface»
Container
+ createIterator(): Iterator

YourContainerClass
+ createIterator(): Iterator

Returns instance of YourIteratorClass.
```
Iterator Design Pattern: Java

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

@interface Iterable<T>
+ iterator(): Iterator<T>
@end

YourIterableClass
+ iterator(): Iterator<T>

Returns an instance of YourIteratorClass.
```
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

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
+setISBN(): void
+compareTo(other: Book): int
+toString(): String

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

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

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