Design Patterns

CSC207 Fall 2017

Computer Science
UNIVERSITY OF TORONTO
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()

for all o: observers: o.update()

ConcreteSubject
- subjectState
+ getState()
+ setState()

ConcreteObserver
- observerState
- subject : ConcreteSubject
+ update()

Observer
+ update()

observerState = subject.getState()
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 boolean hasObservers() {
        return observers != null && observers.size() > 0;
    }

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

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

    public void notifyObservers() {
        onChanged();
    }

    public void notifyObservers(Object arg) {
        onChanged();
    }
}

class YourObservableClass {
    private State state;

    public State getState() {
        return state;
    }

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

class Observer {
    public void update(Observable o, Object arg) {
        // Implementation
    }
}
```
Observer: Example in Java
Uses of Observer in Java

In reality, people usually implement their own

- Usually can’t or don’t want to subclass 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
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

- Strategy
- contextInterface()

- Strategy
  + algorithmInterface()

- ConcreteStrategyA
  + algorithmInterface()

- ConcreteStrategyB
  + algorithmInterface()
Example: without the Strategy pattern

```java
public class Author {
    private String name;
    private List<Book> books;

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

    public String getName() {
        return name;
    }

    public void addBook(Book book) {
        books.add(book);
    }

    public String toString() {
        return name;
    }
}

public class Book {
    private String name;
    private String isbn;

    public Book(String name, String isbn) {
        this.name = name;
        this.isbn = isbn;
    }

    public String getName() {
        return name;
    }

    public String getISBN() {
        return isbn;
    }

    public void addBook(Book book) {
        books.add(book);
    }

    public String toString() {
        return name;
    }
}
```
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
```
Factory Design Pattern

Problem:

- A class cannot anticipate the class of the object it must create.
- A class wants its subclasses to specify the object it creates.
- Classes delegate responsibility to one of several helper subclasses and you want to localize knowledge of which helper subclass is the delegate.
When to use Factory Method Design Pattern

• One class wants to interact with many possible related objects.

• We want to obscure the creation process for these related objects.

• At a later date, we might want to change the types of the objects we are creating.
Factory Method: A Standard Solution

- **Product**
- **ConcreteProduct**

- **Creator**
  - `FactoryMethod()`
  - `AnOperation()`

- **ConcreteCreator**
  - `FactoryMethod()`

- `product = FactoryMethod()`
- `return new ConcreteProduct`
Factory Pattern: An example
Code for this example can be found here:

https://www.tutorialspoint.com/design_pattern/factory_pattern.htm
We can easily create new subclasses of Shape and use the ShapeFactory to instantiate them without changing the rest of the code.
Factory Pattern: Another Example

```
Document
Open()
Close()
Save()
Revert()

MyDocument

Application
CreateDocument()
NewDocument()
OpenDocument()

MyApplication
CreateDocument()

Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();

return new MyDocument
```
For implementation of the Document Example, see:

Model-View-Controller (MVC) Design Pattern

• Model: data, rules that govern access to the data, interactions that update the data
  • Often approximates a real-world process

• View: presentation of that data that is responsive to changes in the model
  • With a Push Model, the View receives change notifications from the Model
  • With a Pull Model, the View is responsible for calling the model and retrieving the most current data.
- Controller: Interaction between User and View are translated by the Controller into actions for the Model to perform.

- The controller can call different aspects of the View (close one window and/or open another)

- Graphic User Interfaces (GUI) often comprise the View, where the Controller is the non-visual front end (e.g., telling buttons what to do when they are clicked), and the Model is the data which the GUI displays and with which the User interacts via the Controller.
MVC Pattern: General Structure

From:
https://www.tonymarston.net/php-mysql/model-view-controller-01.png
MVC: Another Representation
MVC: A Solution in Java
Code for this example can be found here:

https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm
Dependency in Object Oriented Programming

- A “dependency” relationship between two classes (also called “using” relationship) means that any change to the second class will change the functionality of the first.

- For example: class AddressBook depends on class Contact because AddressBook contains instances of Contact.

- Other examples of dependencies: loggers, handlers, listeners
Dependencies in Java

• Using the `new` operator inside the first class can create an instance of a second class that cannot be used nor tested independently. This is called a “hard dependency”.

• We often want to use and test the second class independently from the first and avoid hard dependencies. For example:

  ```java
  public class MyClass {
      private Student student;

      public MyClass(Student student) {
          this.student = student;
      }
  }
  ```
Dependency Injection Design Pattern

Problem:

• We do not want to gather all arguments to call a constructor in our current class. Instead we would rather have another class contain this information (called the “dependency container”) and instantiate our object elsewhere.

• We want a service to provide the object we want. For example, our ShapeFactory created instances of Shape, but the objects were instantiated in the main() method. Instead, we could have written a class called ShapeCreationService that returns instances of Shape for us.
Dependency Injection Example

```
interface MovieFinder

findAll(): List<Movie>
```

```
Movie

director: String
...

Movie(director: String) {create}
getDirector(): String
...
```

```
SemiColonDelimitedMovieFinder

filename: String

setFilename(filename: String): void
```

```
Movies

finder: MovieFinder

Movies() {create}
moviesDirectedBy(director: String): List<Movie>
setFinder(MovieFinder finder): void
```
• Notice how the constructor for class Movies takes an argument of type MovieFinder.

• Any change to the Movie class will affect the MovieFinder, but not Movies directly.

• A dependency has been created between Movie and MovieFinder.
Code for this example can be found here:

Dependency Injection Pattern Exercise

- Consider our previous example of class Person and its subclass Student.

- Create a class called Email that contains instance variables of type String called: to, from, cc, subject, and message. It should also contain PrintToScreen() and PrintToFile() methods.

- Class Person should have the ability to send an Email object to another Person object. But that will require our first instance of Person to obtain the second Person object’s name or email address.
• Without Dependency Injection:
  
  • The sender object will have to use reflection to gather their own name/email address, and somehow obtain the receiver object’s name/email address, etc.
  
  • The code for creating and sending an email will have to be inside class `Person`.
  
  • Any change to the `Email` class will impact the code in class `Person`.
• With Dependency Injection:

• We can create a class called `EmailService` that gathers the information required to compose and send `Email` objects.

• The constructor for `EmailService` can take in the entire `Person` sender object and `Person` receiver object and retrieve the values of their respective `name` and `email` fields, in order to create a new instance of `Email`.

• Any change to the `Email` class will be contained by the `EmailService` and shielded from class `Person`. 