Design Patterns

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

armahmood786@yahoo.com

alphapeeler.sf.net/pubkeys/pkey.htm

pk.linkedin.com/in/armahmood

www.twitter.com/alphapeeler

www.facebook.com/alphapeeler

abdulmahmood-sss   alphasecure

armahmood786@hotmail.com

http://alphapeeler.sf.net/me

alphasecure@gmail.com

http://alphapeeler.sourceforge.net

http://alphapeeler.tumblr.com

armahmood786@jabber.org

alphapeeler@aim.com

mahmood_cubix   48660186

alphapeeler@icloud.com

http://alphapeeler.sf.net/acms/

Template Method Pattern

Intent

- Define the <u>skeleton of an algorithm</u> in an operation, <u>deferring some steps in subclasses</u>. Template Method <u>lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.</u>

- <u>Base class</u> declares algorithm '<u>placeholders' / 'hooks',</u> and derived classes implement the placeholders.

- In Template pattern, an <u>abstract class exposes defined template(s) to execute its methods</u>. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class. This pattern comes under <u>behavior pattern category</u>.

Problem

- Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

Structure
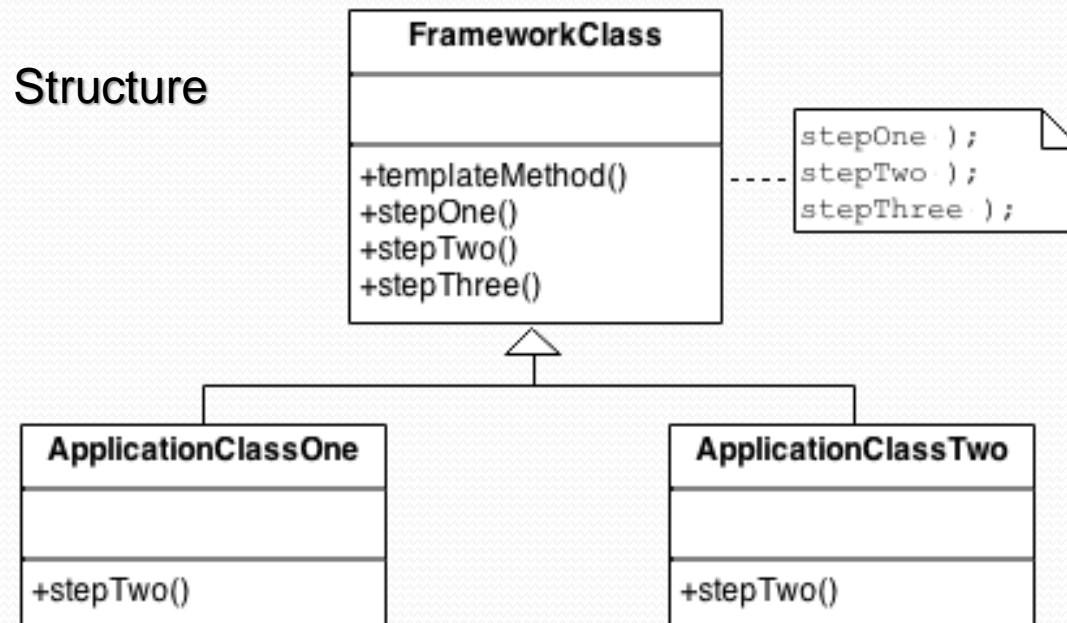
Discussion

- Decide which steps of algorithm are invariant / standard, and which are variant / customizable.

- The invariant steps are implemented in an abstract base class known as "hooks", or "placeholders", that must, be supplied by the component's client in a concrete derived class.

- Designer sets the ordering of required steps of an algorithm, but allows the component client to extend or replace some number of these steps.

- Template Method is used prominently in frameworks. Each framework implements the invariant pieces of a domain's architecture, and defines "placeholders"

Example 1:
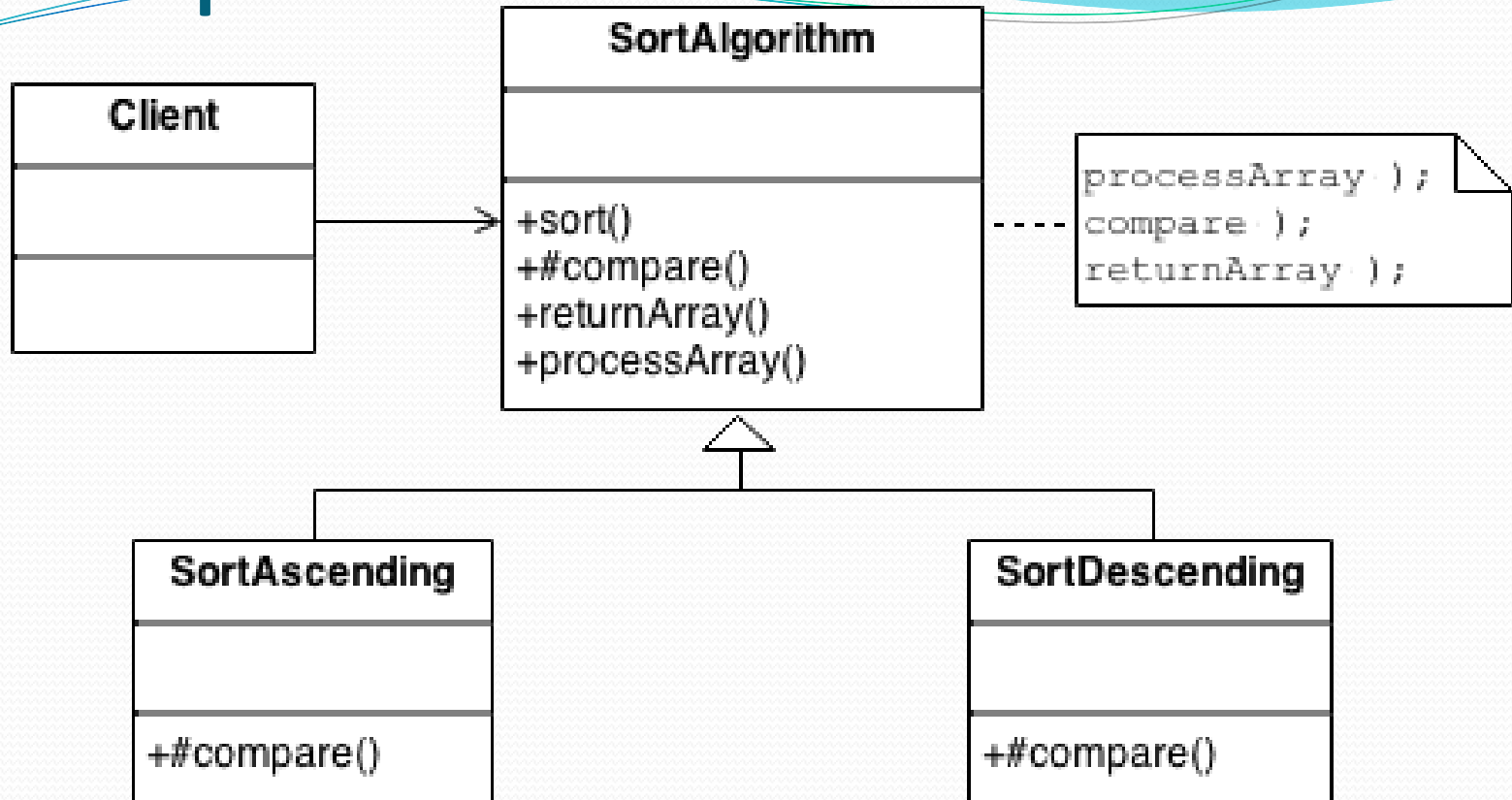
All reusable code is defined in the framework's base classes, and then clients of the framework are free to define customizations by creating derived classes as needed.

Example 2

Worker

+DailyRoutine()
+getUp()
+eatBreakfast()
+goToWork()
+work()
+returnToHome()
+relax()
+sleep()

All workers have
the same daily routine.

Sub-classes override
existing methods of the
template class.

FireFighter

+work()

Lumberjack

+work()

Postman

+work()

Manager

+work()
+relax()
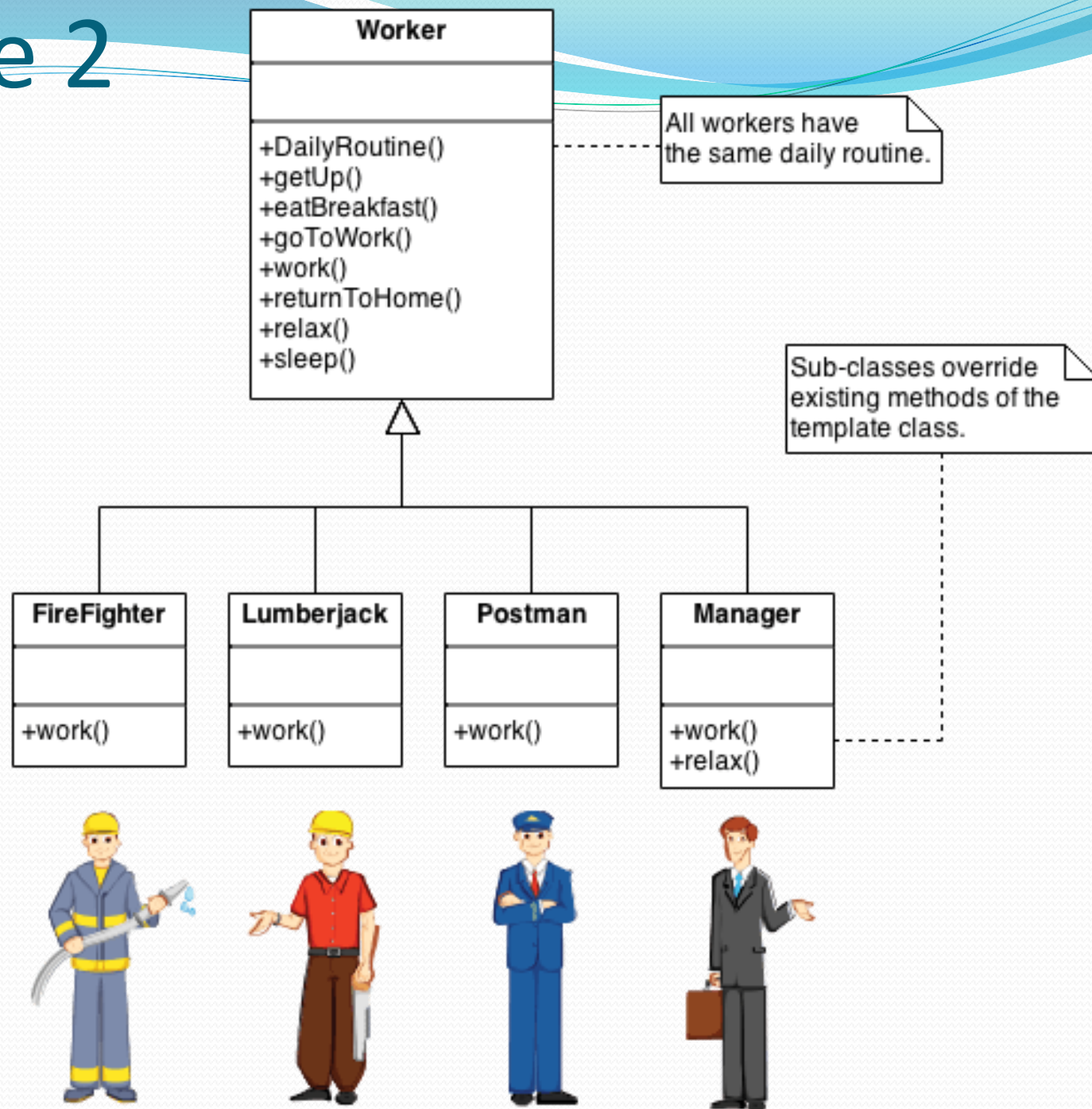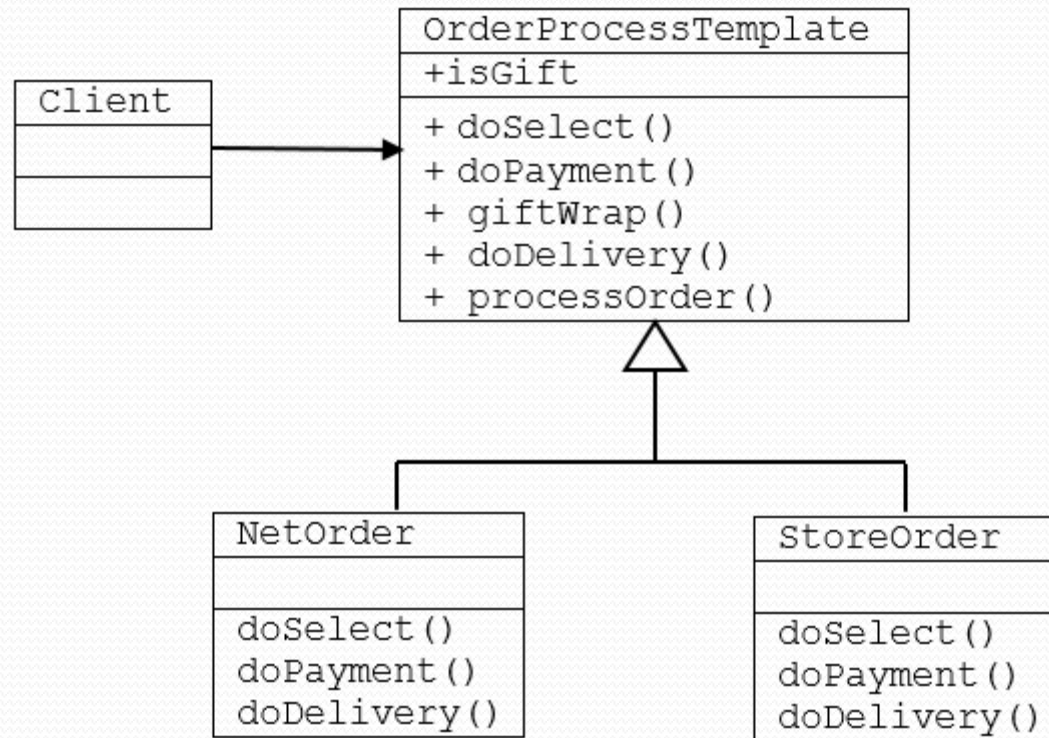
Example 3

Example 3

- Step 1: Create an abstract class with template methods set to be final so that it cannot be overridden.

```java
public abstract class OrderProcessTemplate {
public boolean isGift;
    public abstract void doSelect();
    public abstract void doPayment();
    public final void giftWrap() {
        System.out.println("Gift wrap successfull");
    }
    public abstract void doDelivery();
    public final void processOrder(boolean isGift) {
        doSelect();
        doPayment();
        if (isGift) {
            giftWrap();
        }
        doDelivery();
    }
}
```

Example 3

- Step 2:

```java
public class NetOrder extends OrderProcessTemplate {
    @Override
    public void doSelect() {
        System.out.println("Item added to online shopping cart");
        System.out.println("Get gift wrap preference");
        System.out.println("Get delivery address.");
    }
    @Override
    public void doPayment() {
        System.out.println("Online Payment via Netbanking, card");
    }
    @Override
    public void doDelivery() {
        System.out.println("Ship item via post to delivery address");
    }
}
```

Example 3

- Step 3:

```java
public class StoreOrder extends OrderProcessTemplate {
    @Override
    public void doSelect() {
        System.out.println("Customer chooses the item from shelf.");
    }
    @Override
    public void doPayment() {
        System.out.println("Pays at counter through cash/POS");
    }
    @Override
    public void doDelivery() {
        System.out.println("Item deliverd to in delivery counter.");
    }
}
```

Example 3

- Step 4:

```java
public class TemplateMethodPatternClient {
    public static void main(String[] args) {
        OrderProcessTemplate netOrder = new NetOrder();
        netOrder.processOrder(true);
        System.out.println();
        OrderProcessTemplate storeOrder = new StoreOrder();
        storeOrder.processOrder(true);
    }
}
```

Output:
Item added to online shopping cart
Get gift wrap preference
Get delivery address.
Online Payment via Netbanking, card
Gift wrap successfull
Ship item via post to delivery address

Customer chooses the item from shelf.
Pays at counter through cash/POS
Gift wrap successfull
Item deliverd to in delivery counter.

Example 4
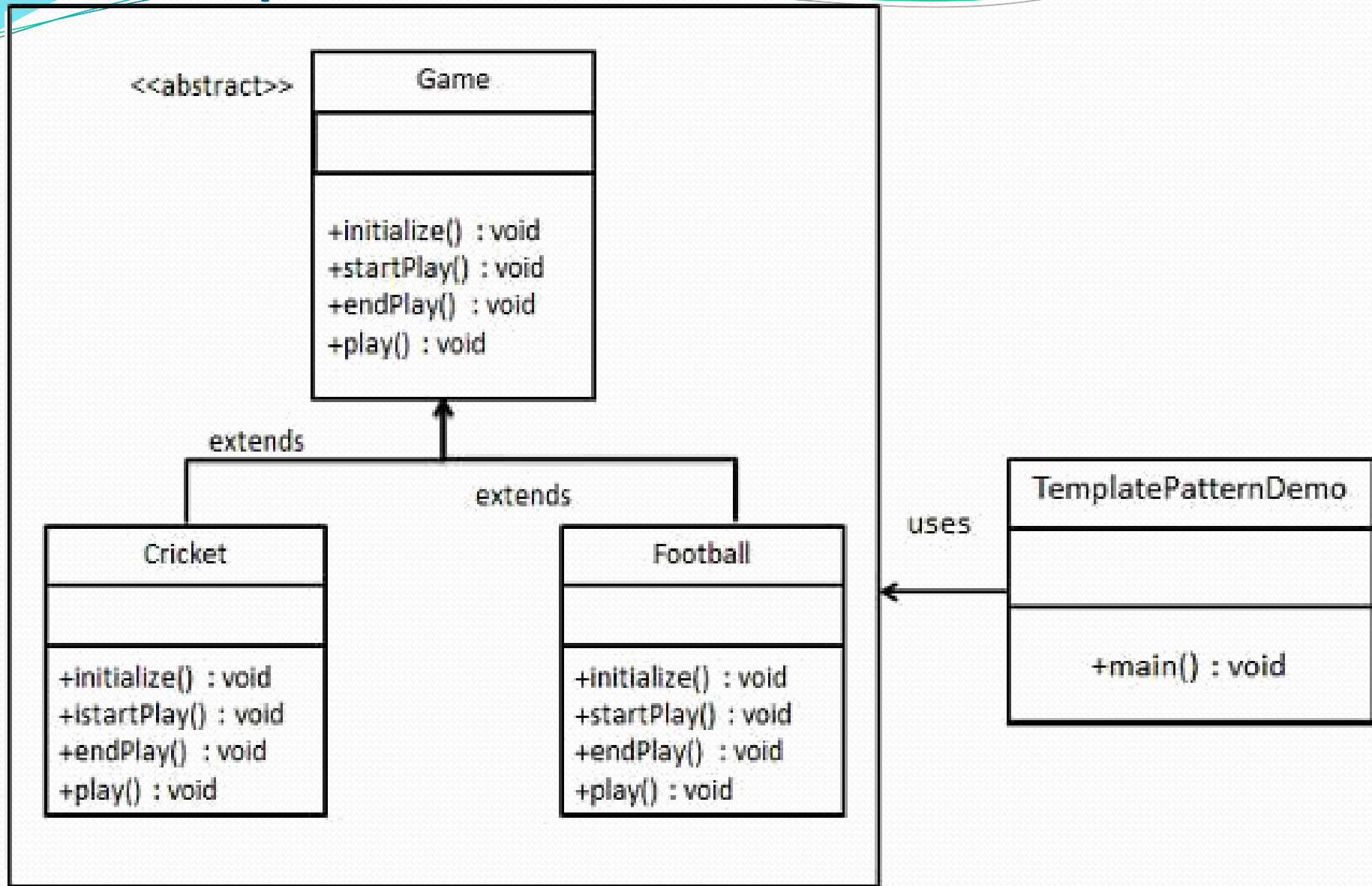
Example 4

- **Step 1:** Step 1: Create an abstract class with template methods set to be final so that it cannot be overridden.

```java
public abstract class Game {
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();

    //template method
    public final void play(){

        //initialize the game
        initialize();

        //start game
        startPlay();

        //end game
        endPlay();
    }
}
```

Example 4

- **Step 2:** Create concrete classes extending above class.
- *Cricket.java*

```java
public class Cricket extends Game {

    @Override
    void endPlay() {
        System.out.println("Cricket Game Finished!");
    }

    @Override
    void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }
}
```

Example 4

- **Step 2:** Create concrete classes extending above class.
- *Football.java*

```java
public class Football extends Game {

    @Override
    void endPlay() {
        System.out.println("Football Game Finished!");
    }

    @Override
    void initialize() {
        System.out.println("Football Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Football Game Started. Enjoy the game!");
    }
}
```

Example 4

- **Step 3:** Use the *Game*'s template method play() to demonstrate a defined way of playing game.

- *TemplatePatternDemo.java*

```java
public class TemplatePatternDemo {
    public static void main(String[] args) {
        Game game = new Cricket();
        game.play();
        System.out.println();
        game = new Football();
        game.play();
    }
}
```

- **Step 4:** Verify the output.

```
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!
```

Iterator Design Pattrn

How to use Iterator in Java?

- 'Iterator' is an interface which belongs to collection framework. It allows us to traverse the collection, access the data element and remove the data elements of the collection.

- **java.util** package has **public interface Iterator** and contains three methods:

- **boolean hasNext**(): It returns true if Iterator has more element to iterate.

- **Object next**(): It returns the next element in the collection until the hasNext()method return true. This method throws 'NoSuchElementException' if there is no next element.

- **void remove**(): It removes the current element in the collection. This method throws 'IllegalStateException' if this function is called before next( ) is invoked.

Ex 5: Java iterator

```java
// Java code to illustrate the use of iterator
import java.io.*;
import java.util.*;
class Test {
        public static void main(String[] args)
        {
                ArrayList<String> list = new ArrayList<String>();
                list.add("A");
                list.add("B");
                list.add("C");
                list.add("D");
                list.add("E");
                // Iterator to traverse the list
                Iterator iterator = list.iterator();
                System.out.println("List elements : ");
                while (iterator.hasNext())
                        System.out.print(iterator.next() + " ");
                System.out.println();
        }
}
```

Output:
List elements :
A B C D E

ListIterator

- 'ListIterator' in Java is an Iterator which allows users to traverse Collection in both direction. It contains the following methods:
- **void add(Object object)**: It inserts object immediately before the element that is returned by the next( ) function.
- **boolean hasNext( )**: It returns true if the list has a next element.
- **boolean hasPrevious( )**: It returns true if the list has a previous element.
- **Object next( )**: It returns the next element of the list. It throws 'NoSuchElementException' if there is no next element in the list.
- **Object previous( )**: It returns the previous element of the list. It throws 'NoSuchElementException' if there is no previous element.
- **void remove( )**: It removes the current element from the list. It throws 'IllegalStateException' if this function is called before next( ) or previous( ) is invoked.

Ex 6: Java iterator

```java
import java.io.*;
import java.util.*;
class Test {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("A"); list.add("B"); list.add("C");
        list.add("D"); list.add("E");
        // ListIterator to traverse the list
        ListIterator iterator = list.listIterator();
        // Traversing the list in forward direction
        System.out.println("List elements - forward : ");
        while (iterator.hasNext())
            System.out.print(iterator.next() + " ");
        System.out.println();
        // Traversing the list in backward direction
        System.out.println("List elements - backward : ");
        while (iterator.hasPrevious())
            System.out.print(iterator.previous() + " ");
        System.out.println();
    }
}
```

```
Output:
List elements - forward :
A B C D E
List elements - backward :
E D C B A
```

IteratorPattern

- See the example document provided you to as Week08c_IteratorPattern-ReadingAssignment.pdf
- Happy reading!