



# M7 (a) - Inheritance

Jin L.C. Guo

*Image source [https://cdn.pixabay.com/photo/2015/01/11/21/30/cats-596782\\_1280.jpg](https://cdn.pixabay.com/photo/2015/01/11/21/30/cats-596782_1280.jpg)*

# Reminder

- Lab 3
  - Starts on this Thursday (Mar 16<sup>th</sup>)
  - Coverage: Design for Robustness, Unit Testing, Composition
- Finish booking Lab 3 and 4

# Recap of Module 6

- Design Principle:

Divide and Conquer, Law of Demeter

- Programming mechanism:

Aggregation and Delegation, Polymorphic Object Cloning

- Design Techniques:

Sequence Diagram

- Patterns and Anti-patterns:

Composite Pattern, Decorator Pattern, Prototype Pattern, Command Pattern, God class

# Objective

- Programming mechanism:

Inheritance, subtyping, downcasting, object initialization, super calls, overriding, overloading, abstract classes, abstract methods, final classes, final methods;

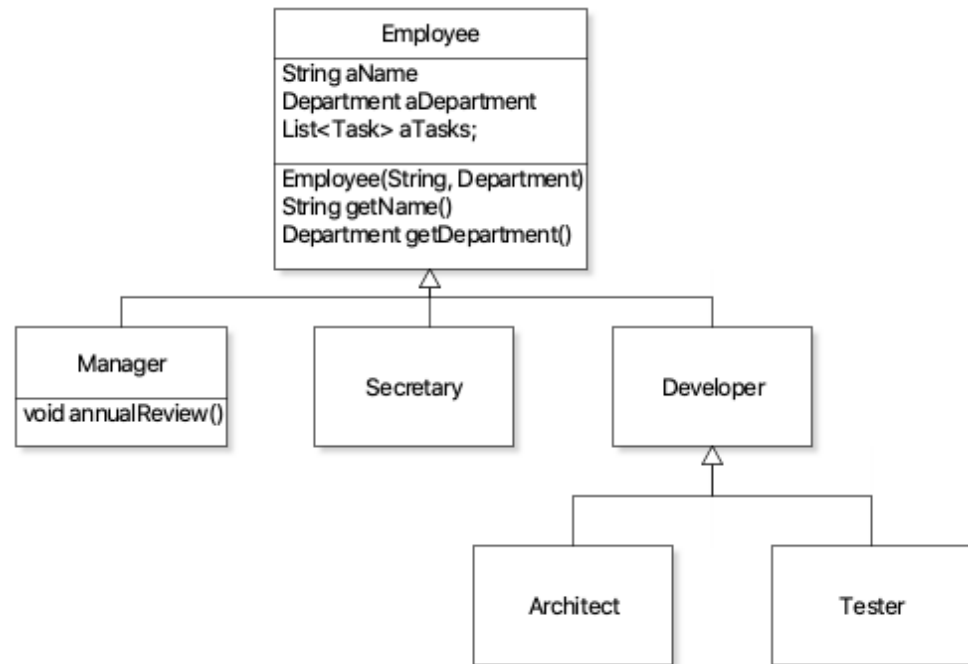
- Design Techniques:

Inheritance-based reuse

- Patterns and Anti-patterns:

Template Pattern

# Inheritance



```
Employee e1, e2;  
e1 = new Developer("July", new Department("Security"));  
e2 = new Manager("Diana", new Department("Security"));
```

# Run-time vs Compile-time Type

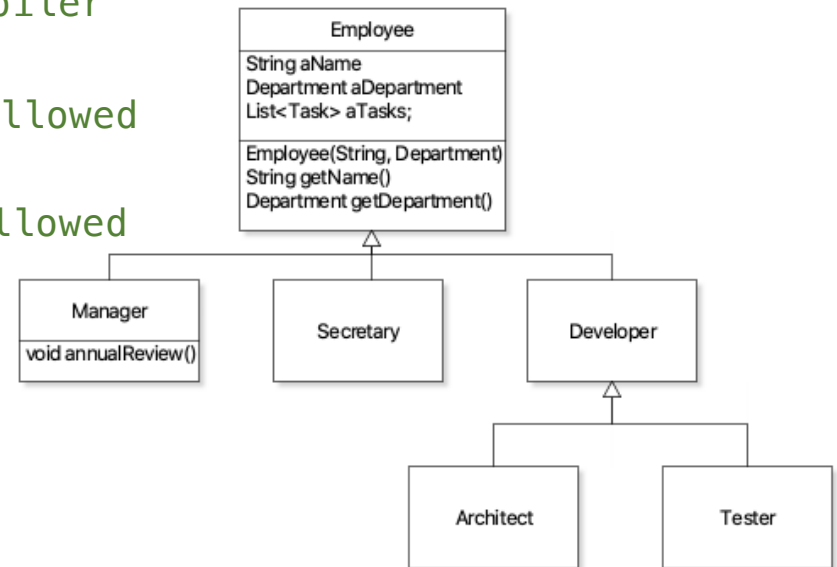
```
Employee e1, e2;  
e1 = new Developer("July", new Department("Security"));  
e2 = new Manager("Diana", new Department("Security"));
```

```
e2.annualReview(); // not allowed by the compiler
```

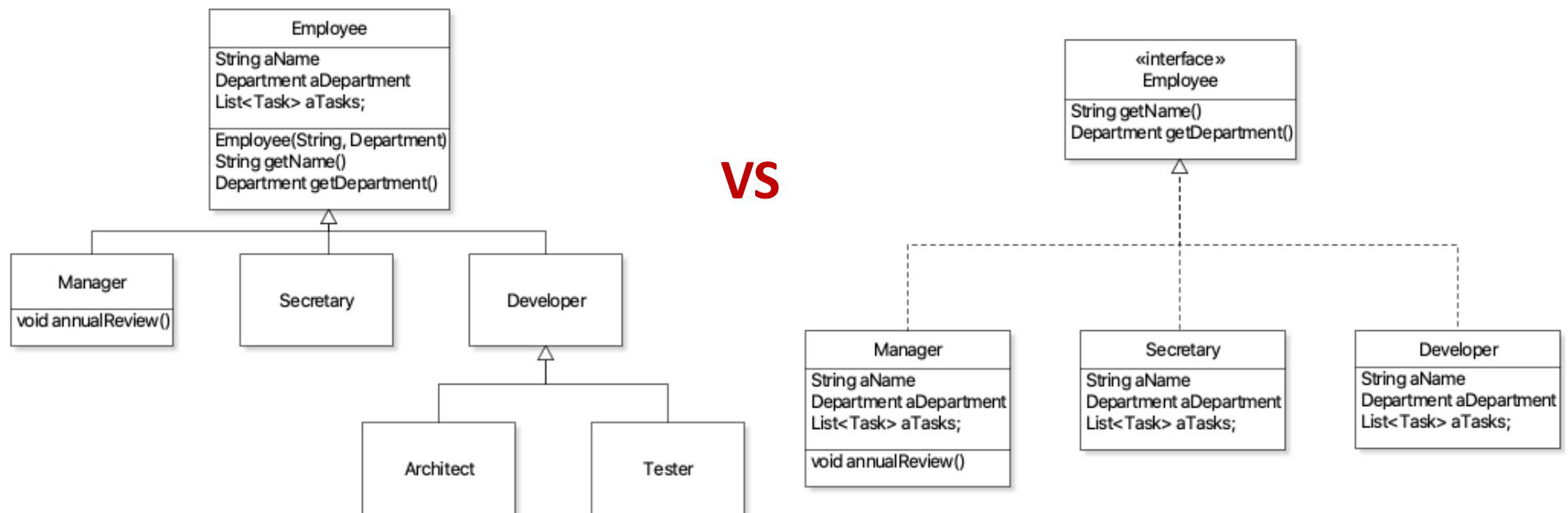
```
((Manager) e2).annualReview(); // compiler allowed
```

```
((Manager) e1).annualReview(); // compiler allowed  
// but run-time exception!
```

```
System.out.println(e1 instanceof Manager);  
System.out.println(e2 instanceof Manager);
```



# Comparing Inheritance and Interface



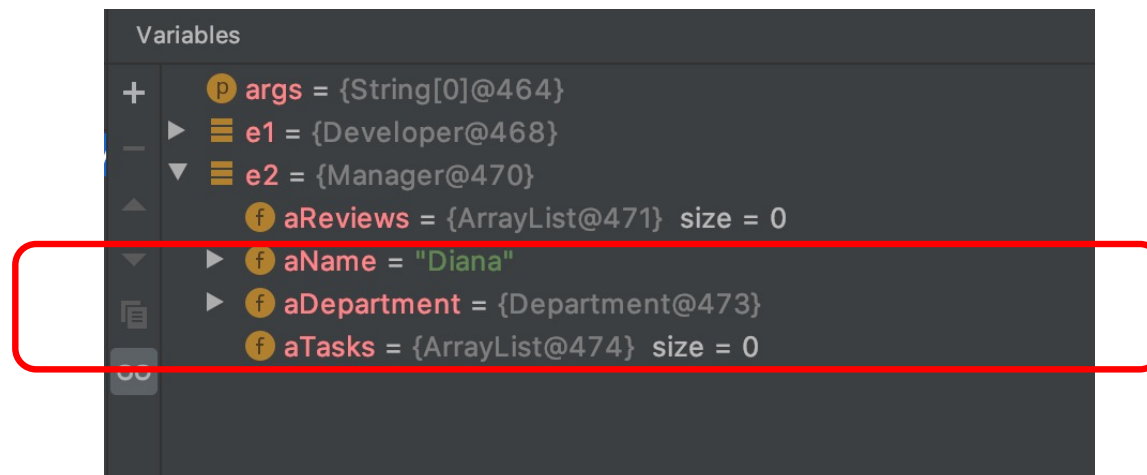
```
public class Employee {  
    private String aName;  
    private Department aDepartment;  
    private List<Task> aTasks = new ArrayList<>();  
  
    Employee(String pName, Department pDepartment) {  
        aName = pName;  
        aDepartment = pDepartment;  
    }  
  
    .....  
}
```

```
public class Manager extends Employee  
{  
    private final List<Review> aReviews = new ArrayList<>();  
}
```



# Inheriting Fields

```
Employee e1, e2;  
e1 = new Developer("July", new Department("Security"));  
e2 = new Manager("Diana", new Department("Security"));
```



# Subclass Constructor

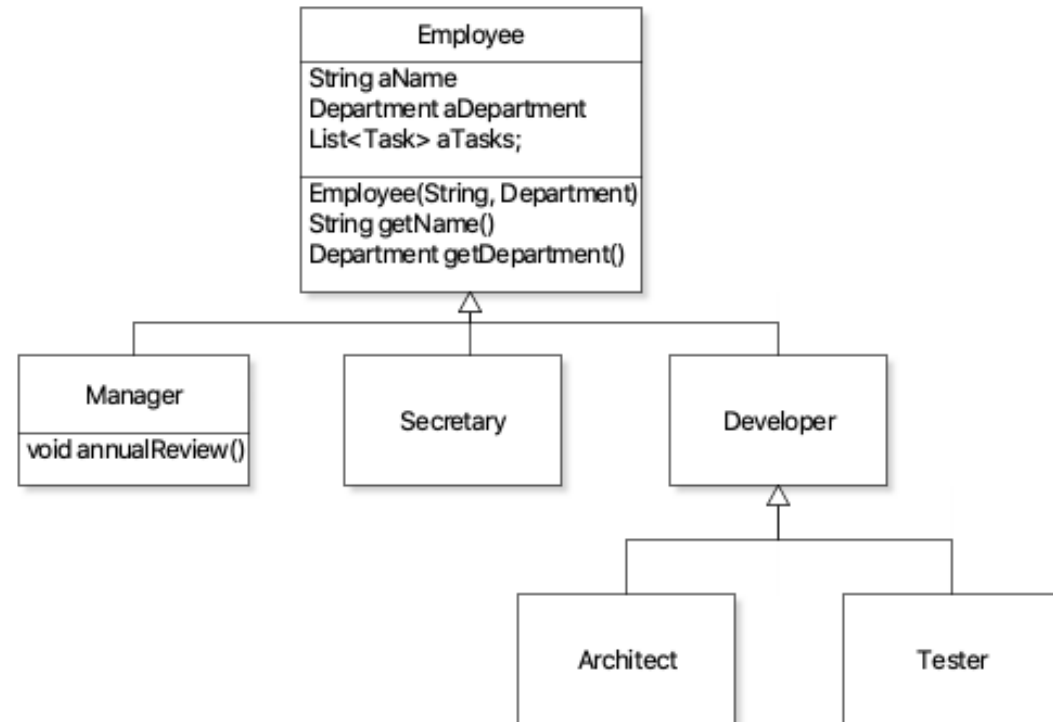
```
public Manager(String pName, Department pDepartment) {  
    aName = pName;  
    aDepartment= pDepartment;  
}
```

# Subclass Constructor

```
public Manager(String pName, Department pDepartment) {  
    super(pName, pDepartment); new Employee(pName, pDepartment);  
}
```

# Inheriting Methods

```
Employee e2 = new Manager("Diana",  
    new Department("Security"));  
e2.getName();
```



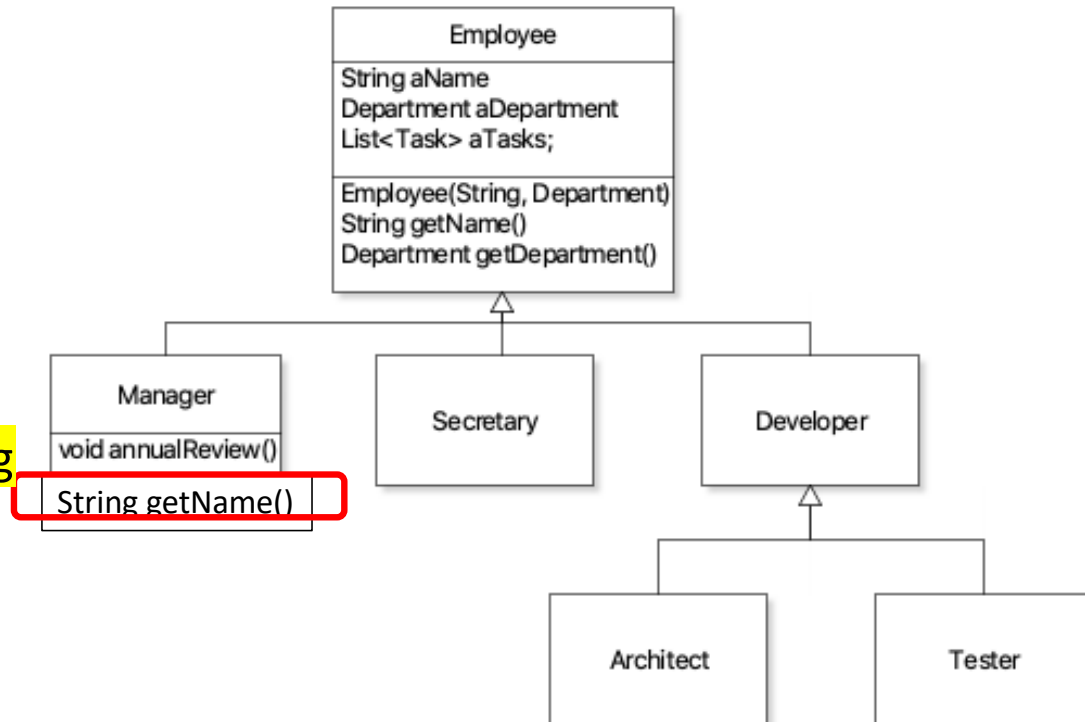
# Override Methods

```
public class Manager extends Employee {  
    private List<Review> aReviews = new ArrayList<>();  
  
    public Manager(String pName, Department pDepartment) {  
        super(pName, pDepartment);  
    }  
  
    @Override  
    public String getName() {  
        return "Manager " + super.getName();  
    }  
}
```

# Inheriting Methods

```
Employee e2 = new Manager("Diana",  
    new Department("Security"));  
e2.getName();
```

Dynamic Binding



# Objective

- Programming mechanism:

Inheritance, subtyping, downcasting, object initialization, super calls, overriding, overloading, abstract classes, abstract methods, final classes, final methods;

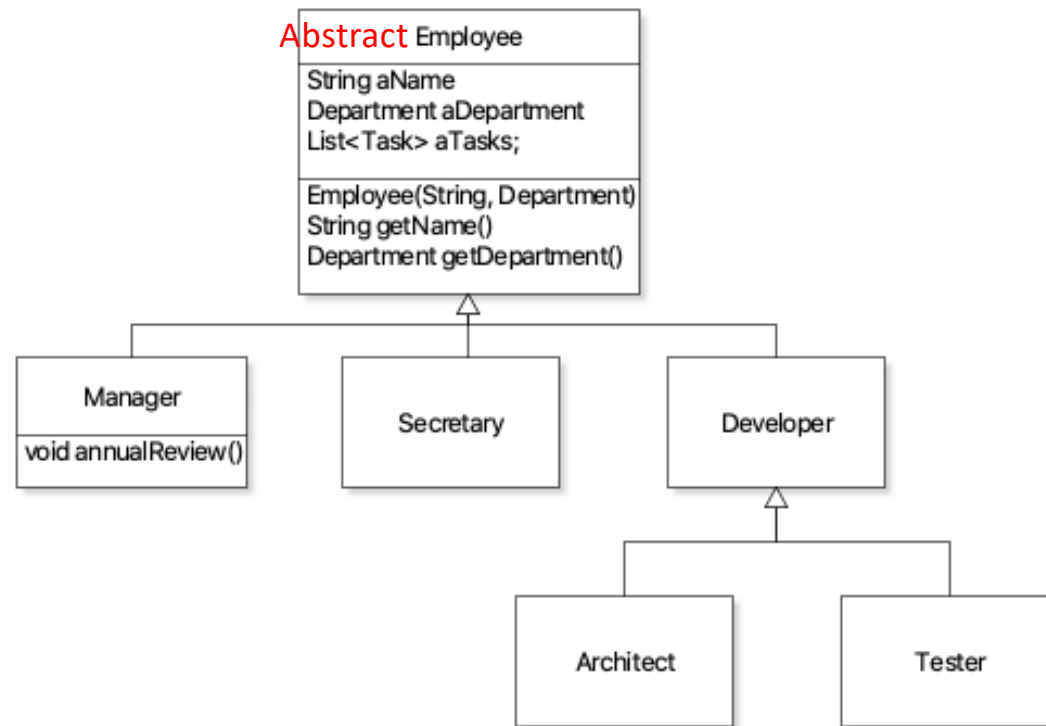
- Design Techniques:

Inheritance-based reuse

- Patterns and Anti-patterns:

Template Pattern

# Abstract Class

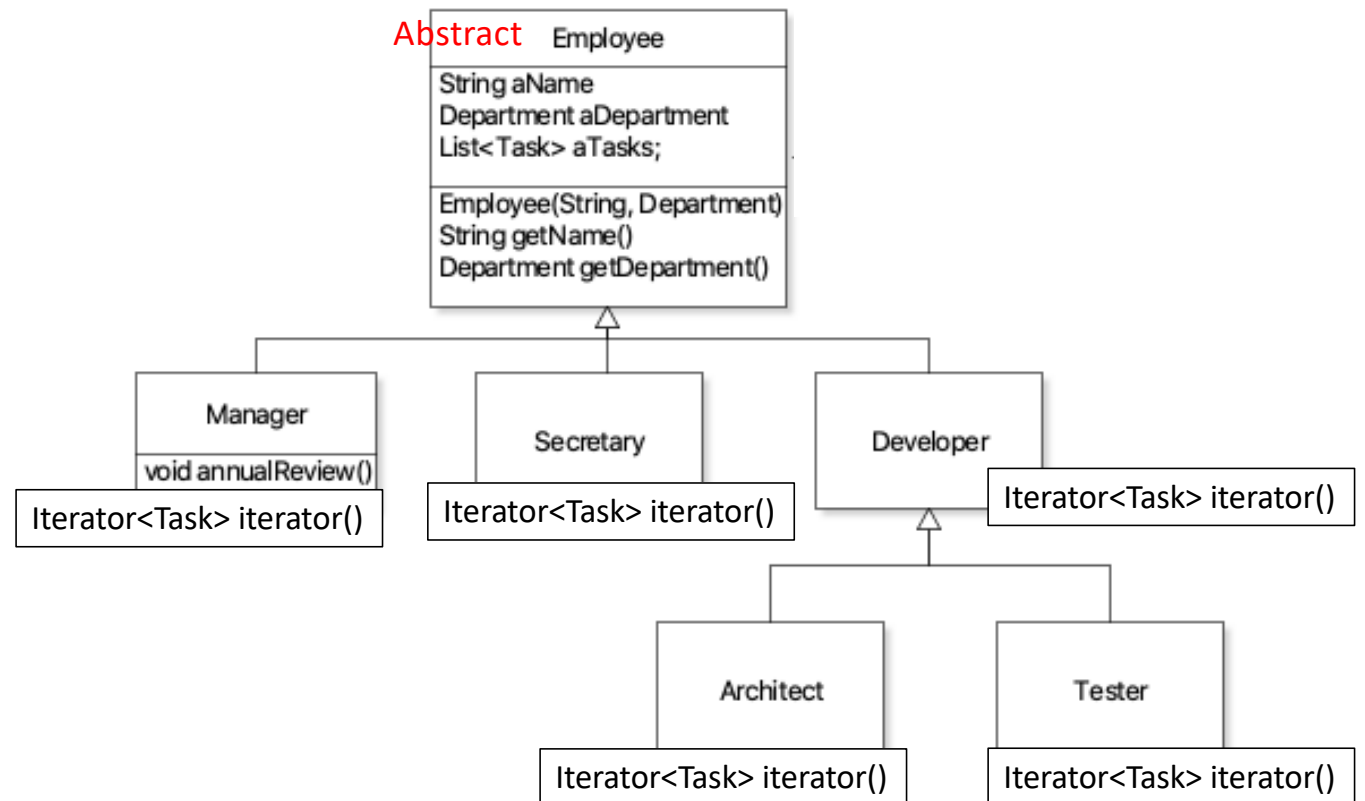




# Abstract Class

- The class cannot be instantiated
- Can declare abstract methods
  - Subclass needs to implement
- No longer needs to supply implementations to all methods in the interface it declares to implement.
  - Subclass needs to implement

# Abstract Class



```
public abstract class Employee implements Iterable<Task>{
    private String aName;
    private Department aDepartment;
    private List<Task> aTasks = new ArrayList<>();

    Employee(String pName, Department pDepartment) {
        aName = pName;
        aDepartment= pDepartment;
    }

    public String getName() {
        return aName;
    }

    public Department getDepartment() {
        return aDepartment;
    }

    public abstract void printNameCard(Printer p);
}
```

# Objective

- Programming mechanism:

Inheritance, subtyping, downcasting, object initialization, super calls, overriding, overloading, abstract classes, abstract methods, final classes, final methods;

- Design Techniques:

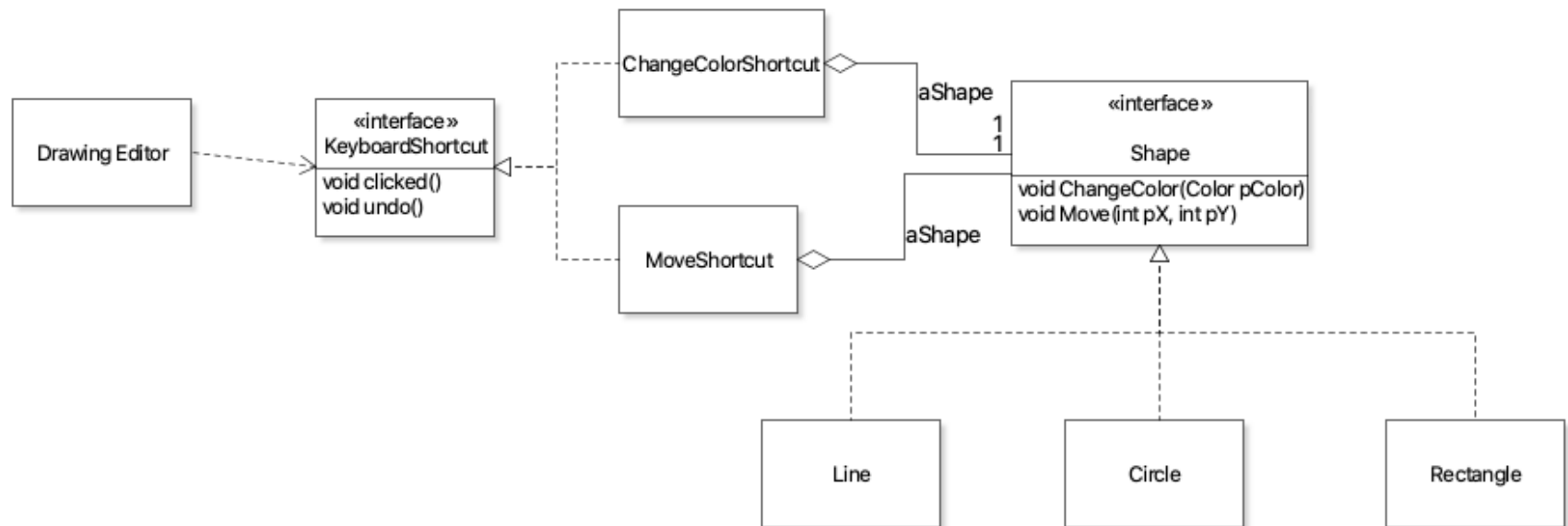
Inheritance-based reuse

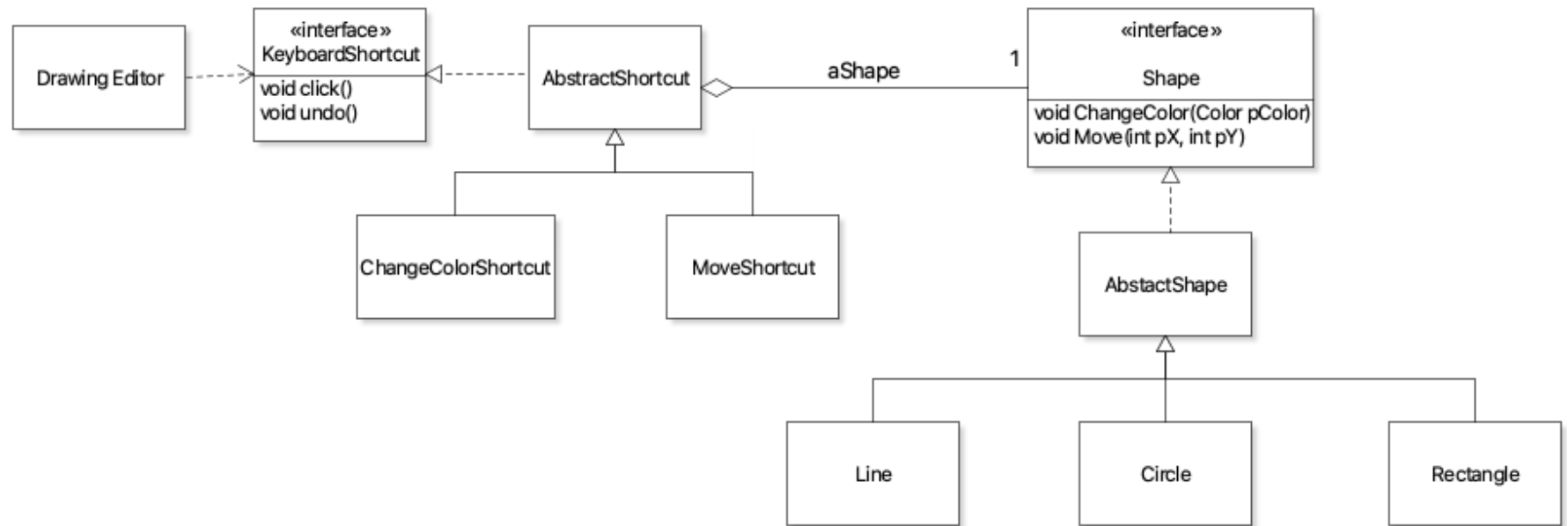
- Patterns and Anti-patterns:

Template Pattern

# Activity1:

- Use inheritance to remove redundancy in the following design of applying command pattern to drawing editor





```
public abstract class Employee implements Iterable<Task>{
    private String aName;
    private Department aDepartment;
    private List<Task> aTasks = new ArrayList<>();

    Employee(String pName, Department pDepartment) {
        aName = pName;
        aDepartment= pDepartment;
    }

    public String getName() {
        return aName;
    }

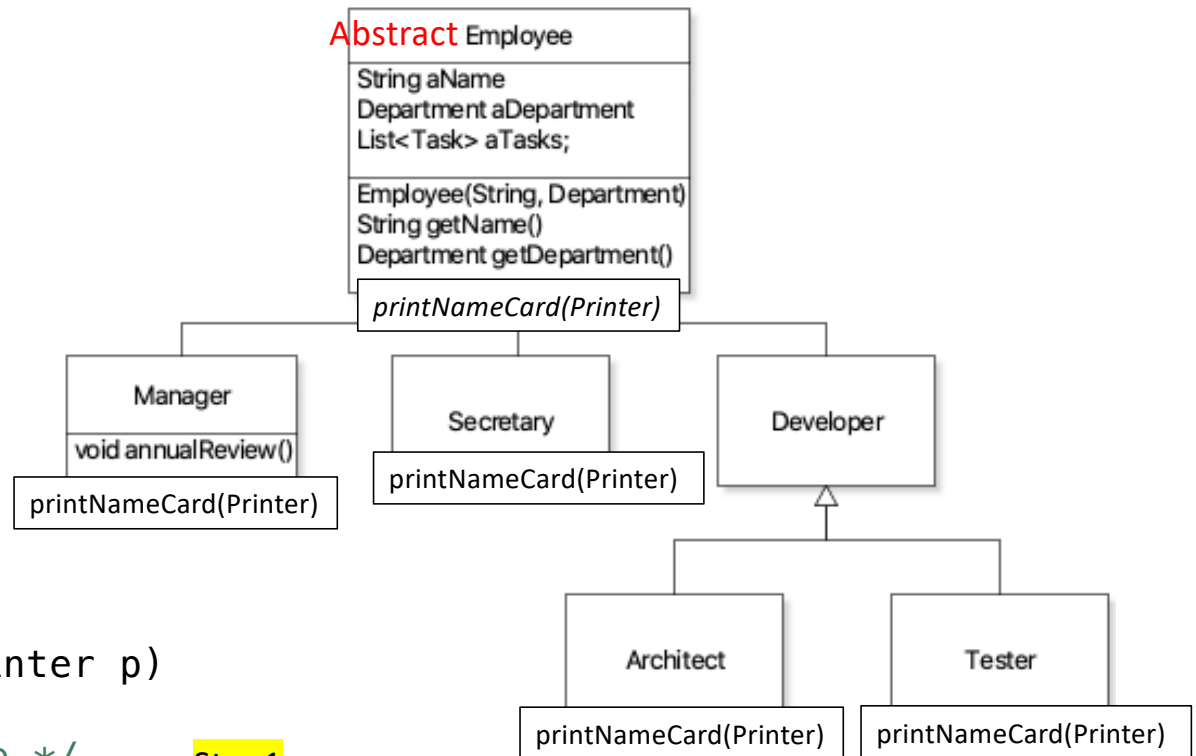
    public Department getDepartment() {
        return aDepartment;
    }

    public abstract void printNameCard(Printer p);
}
```

What if only part of a  
method needs to be  
deferred to the subclass?

---



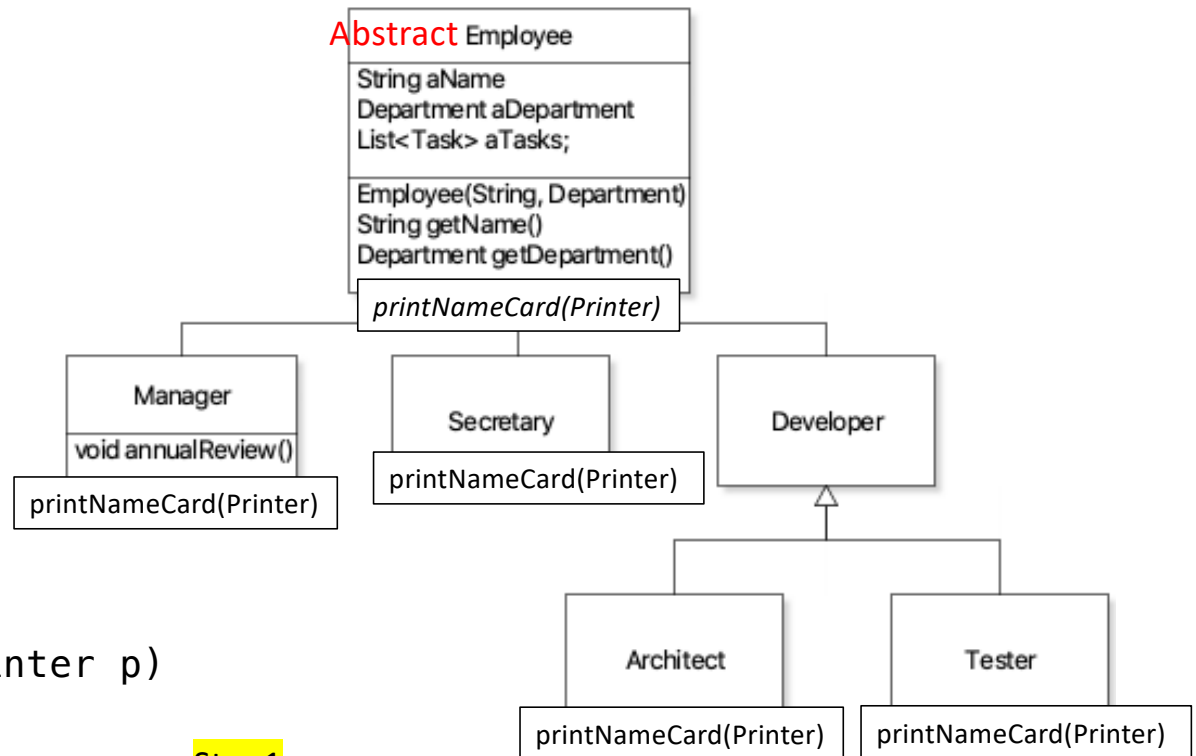


```
public void printNameCard(Printer p)
{
    /* print company template */
    /* print customized content*/
    /* print receipt */
}
```

Step1

Step2

Step3



```

public void printNameCard(Printer p)
{
    p.printTemplate();
    /* print customized content*/
    p.printReceipt();
}
  
```

Step1

Step2

Step3

# A multi-step method

```
public abstract class Employee implements Iterable<Task>{  
    private String aName;  
    private Department aDepartment;  
    private List<Task> aTasks = new ArrayList<>();
```

**final**

Subclass cannot override this method

```
    public void printNameCard(Printer p)  
    {  
        p.printTemplate();  
        p.print(getPrintContent());  
        p.printReceipt();  
    }
```

Step1

Step2

Step3

```
    public abstract String getPrintContent();
```

```
}
```

```
public class Manager extends Employee {  
    private List<Review> aReviews = new ArrayList<>();  
  
    public Manager(String pName, Department pDepartment) {  
        super(pName, pDepartment);  
    }  
  
    @Override  
    public String getPrintContent() {  
        return getName() + getDepartment().toString();  
    }  
}
```

# Objective

- Programming mechanism:

Inheritance, subtyping, downcasting, object initialization, super calls, overriding, overloading, abstract classes, abstract methods, final classes, final methods;

- Design Techniques:

Inheritance-based reuse

- Patterns and Anti-patterns:


Template Pattern

# Template Method Pattern

- Intent:
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Participants:
  - AbstractClass
    - implements a template method defining the skeleton of an algorithm*
    - defines abstract operations that concrete subclasses define to implement steps of an algorithm.*
  - ConcreteClass
    - implements the operations to carry out subclass-specific steps of the algorithm.*

# Template Method Pattern

```
public abstract class Employee implements Iterable<Task>{  
    private String aName;  
    private Department aDepartment;  
    private List<Task> aTasks = new ArrayList<>();
```

```
     final  
    public void printNameCard(Printer p)  
    {  
        p.printTemplate();  
        p.print(getPrintContent());  
        p.printReceipt();  
    }
```

Abstract step method

Avoid same names for template and abstract methods

```
    public abstract String getPrintContent();
```

Protected or Public

```
}
```

Not necessarily abstract (define default behavior)

# Examples of Abstract classes and Template Method Pattern in Java

- [java.util.AbstractList](#)
- [java.util.AbstractSet](#)
- [java.util.AbstractMap](#)
- [java.io.InputStream](#)
- [java.io.OutputStream](#)
- [java.io.Reader](#)
- [java.io.Writer](#)

... ..



## java.util.AbstractList

- Implemented Interfaces:

Iterable<E>, Collection<E>, List<E>

- Direct Subclasses:

AbstractSequentialList, ArrayList, Vector

## java.util.AbstractList

This class provides a skeletal implementation of the [List](#) interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).

To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the [get\(int\)](#) and [size\(\)](#) methods.

To implement a modifiable list, the programmer must additionally override the `set(int, E)` method (which otherwise throws an `UnsupportedOperationException`).

... ..

```
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E>
{
    ... ..
    abstract public E get(int index);

    public E next() {
        checkForComodification();
        try {
            int i = cursor;
            E next = get(i);
            lastRet = i;
            cursor = i + 1;
            return next;
        } catch (IndexOutOfBoundsException e) {
            checkForComodification();
            throw new NoSuchElementException();
        }
    }
    ... ..
}
```

## Activity 2:

- Using inheritance to design a class representing an unmodifiable list of Card that is constructed through a card array.
- What methods do you need to override?
- How to override them?

```
public class CardList extends AbstractList<Card>
{
    private final Card[] aCards;

    CardList(Card[] pCards)
    {
        assert pCards != null;
        aCards = pCards;
    }

}
```

```
public static void main(String[] pArgs)
{
    Card[] cards = new Card[2];
    cards[0] = new Card(Rank.ACE, Suit.CLUBS);
    cards[1] = new Card(Rank.FIVE, Suit.DIAMONDS);
    CardList cardList = new CardList(cards);

    System.out.println(cardList.contains(cards[1]));

    for (Iterator<Card> iter=cardList.iterator();
         iter.hasNext(); )
    {
        Card element = iter.next();
        System.out.println(element);
    }
}
```