the State Pattern
the State Pattern
the State Pattern

what if a given method should behave differently depending on some internal state of the object?
the State Pattern

what if a given method should behave differently depending on some internal state of the object?
the State Pattern

what if a given method should behave differently depending on some internal state of the object?

“Hi honey, I’m home. What’s for din-din?”
the State Pattern

what if a given method should behave differently depending on some internal state of the object?

“Hi honey, I’m home. What’s for din-din?”
the State Pattern

what if a given method should behave differently depending on some internal state of the object?

“Hi honey, I’m home. What’s for din-din?”

might return different responses dependent upon the state of the relationship, or the state of mind of the partner.
the State Pattern

what if a given method should behave differently depending on some internal state of the object?

“Hi honey, I’m home. What’s for din-din?”

might return different responses dependent upon the state of the relationship, or the state of mind of the partner.
the State Pattern

what if a given method should behave differently depending on some internal state of the object?

“Hi honey, I’m home. What’s for din-din?”

might return different responses dependent upon the state of the relationship, or the state of mind of the partner.

Same partner but different behaviors.
the State Pattern

what if a given method should behave differently depending on some internal state of the object?

“Hi honey, I’m home. What’s for din-din?”

might return different responses dependent upon the state of the relationship, or the state of mind of the partner.

Same partner but different behaviors.
state-dependent methods
state-dependent methods

basic mechanism of the pattern:
state-dependent methods

basic mechanism of the pattern:
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:
state-dependent methods

basic mechanism of the pattern:

as in the **Strategy Pattern**, delegate, or hand off, the task to some other class better able to perform the task:
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:

class A {
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();
}
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();
}
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
}
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:

class A {
  private Delegate d = new Delegate();

  public void m1() { d.m1(); }
  public void m2() { d.m2(); }
}
state-dependent methods

basic mechanism of the pattern:

as in the **Strategy Pattern**, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

Friday, May 14, 2010
state-dependent methods

basic mechanism of the pattern:

as in the **Strategy Pattern**, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states
state-dependent methods

basic mechanism of the pattern:

as in the **Strategy Pattern**, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states
state-dependent methods

basic mechanism of the pattern:

as in the **Strategy Pattern**, delegate, or hand off, the task to some other class better able to perform the task:

```java
class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}
```

So think of people as having multiple states

```java
class Partner {
```
state-dependent methods

basic mechanism of the pattern:

as in the **Strategy Pattern**, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states

class Partner {
    private GrumpyState grumpyState = new GrumpyState();
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states

class Partner {
    private GrumpyState grumpyState = new GrumpyState();
    private HappyState happyState = new HappyState();
}
state-dependent methods

basic mechanism of the pattern:

as in the **Strategy Pattern**, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states

class Partner {
    private GrumpyState grumpyState = new GrumpyState();
    private HappyState happyState = new HappyState();
}
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states

class Partner {
    private GrumpyState grumpyState = new GrumpyState();
    private HappyState happyState = new HappyState();

    public void whatz4DinDin() {

```
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task
to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states

class Partner {
    private GrumpyState grumpyState = new GrumpyState();
    private HappyState happyState = new HappyState();

    public void whatz4DinDin() {
        if (...) grumpyState.m1() else happyState.m1(); }

state-dependent methods

basic mechanism of the pattern:

as in the **Strategy Pattern**, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states

class Partner {
    private GrumpyState grumpyState = new GrumpyState();
    private HappyState happyState = new HappyState();

    public void whatz4DinDin() {
        if (...) grumpyState.m1() else happyState.m1();
    }
}
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task
to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states

class Partner {
    private GrumpyState grumpyState = new GrumpyState();
    private HappyState happyState = new HappyState();

    public void whatz4DinDin() {
        if (...) grumpyState.m1() else happyState.m1();
    }
}
state-dependent methods

basic mechanism of the pattern:

as in the Strategy Pattern, delegate, or hand off, the task to some other class better able to perform the task:

class A {
    private Delegate d = new Delegate();

    public void m1() { d.m1(); }
    public void m2() { d.m2(); }
}

So think of people as having multiple states

class Partner {
    private GrumpyState grumpyState = new GrumpyState();
    private HappyState happyState = new HappyState();

    public void whatz4DinDin() {
        if (...) grumpyState.m1() else happyState.m1();
    }
}
adding an abstraction layer
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();
}
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
}
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }
}
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }

    public boolean negate(boolean v) { return !v; }
}
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }

    public boolean negate(boolean v) { return !v; }
    public int addOne(int i) { return i+1; }
}
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }

    public boolean negate(boolean v) { return !v; }
    public int addOne(int i) { return i+1; }
}
adding an abstraction layer

we need to look over our state-dependent class, deciding which methods are state-dependent and which are not.

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }

    public boolean negate(boolean v) { return !v; }
    public int addOne(int i) { return i+1; }
}
how to call get these methods to depend on g versus h?
how to call get these methods to depend on g versus h?
how to call get these methods to depend on g versus h?
class Partner {

how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
}
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }

    ANS: create an interface or an abstract class to define which methods are state dependent:
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }

    ANS: create an interface or an abstract class to define which methods are state dependent:
how to call get these methods to depend on g versus h?

class Partner {
   private GrumpyState g = new GrumpyState();
   private HappyState h = new HappyState();

   public void whatz4DinDin() { depends on g versus h }
   public void wannaGo2PDX() { depends on g versus h }

   ANS: create an interface or an abstract class to define which methods are state dependent:

   abstract public class PartnerState {

how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState  h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }

ANS: create an interface or an abstract class to define which methods are state dependent:

abstract public class PartnerState {
    abstract public void whatz4DinDin();
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }

    ANS: create an interface or an abstract class to define which methods are state dependent:

    abstract public class PartnerState {
        abstract public void whatz4DinDin();
        abstract public void wannaGo2PDX();
    }
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }
}

ANS: create an interface or an abstract class to define which methods are state dependent:

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }
}

ANS: create an interface or an abstract class to define which methods are state dependent:

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }
}

ANS: create an interface or an abstract class to define which methods are state dependent:

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState { ...
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }
}

ANS: create an interface or an abstract class to define which methods are state dependent:

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

class GrumpyState extends PartnerState { ... 
public class HappyState extends PartnerState { ...
how to call get these methods to depend on g versus h?

class Partner {
    private GrumpyState g = new GrumpyState();
    private HappyState h = new HappyState();

    public void whatz4DinDin() { depends on g versus h }
    public void wannaGo2PDX() { depends on g versus h }

    ANS: create an interface or an abstract class to define which methods are state dependent:

    abstract public class PartnerState {
        abstract public void whatz4DinDin();
        abstract public void wannaGo2PDX();
    }

    public class GrumpyState extends PartnerState { ... }
    public class HappyState extends PartnerState { ... }
The State Hierarchy
The State Hierarchy

abstract public class PartnerState {

The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {

The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {

The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {

    public void whatz4DinDin() { System.err.println("road kill."); }
}
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {

    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}
The State Hierarchy

abstract public class PartnerState {
  abstract public void whatz4DinDin();
  abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {

  public void whatz4DinDin() { System.err.println("road kill."); }
  public void wannaGo2PDX() { System.err.println("no"); }
}
The State Hierarchy

abstract public class PartnerState {
   abstract public void whatz4DinDin();
   abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
   public void whatz4DinDin() { System.err.println("road kill."); }
   public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState { ...
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {

    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState { ...}
abstract public class PartnerState {
  abstract public void whatz4DinDin();
  abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
  public void whatz4DinDin() { System.err.println("road kill."); }
  public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState { ...
  public void whatz4DinDin() { System.err.println("chicken"); }
}
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX() { System.err.println("yeah!!!"); }
}
abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX()  { System.err.println("no"); }
}

public class HappyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX()  { System.err.println("yeah!!!"); }
}
abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX() { System.err.println("yeah!!!"); }
}
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {

    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX()  { System.err.println("no"); }
}

public class HappyState extends PartnerState {

    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX()  { System.err.println(“yeah!!!”); }
}

public class SleepState extends PartnerState {

The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState { ...
    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX() { System.err.println("yeah！！！"); }
}

public class SleepState extends PartnerState {
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX() { System.err.println("yeah!!!"); }
}

public class SleepState extends PartnerState {
    public void whatz4DinDin() { System.err.println("mumble"); }
}
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX() { System.err.println("yeah!!!"); }
}

public class SleepState extends PartnerState {
    public void whatz4DinDin() { System.err.println("mumble"); }
    public void wannaGo2PDX() { System.err.println("mummph"); }
}
abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX() { System.err.println("yeah!!!"); }
}

public class SleepState extends PartnerState {
    public void whatz4DinDin() { System.err.println("mumble"); }
    public void wannaGo2PDX() { System.err.println("mummph"); }
}
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public void whatz4DinDin() { System.err.println("road kill."); }
    public void wannaGo2PDX() { System.err.println("no"); }
}

public class HappyState extends PartnerState { ...
    public void whatz4DinDin() { System.err.println("chicken"); }
    public void wannaGo2PDX() { System.err.println("yeah!!!"); }
}

public class SleepState extends PartnerState {
    public void whatz4DinDin() { System.err.println("mumble"); }
    public void wannaGo2PDX() { System.err.println("mummph"); }
}
The State Hierarchy

abstract public class PartnerState {
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {

    public void whatz4DinDin() {
        System.err.println("road kill.");
    }
    public void wannaGo2PDX() {
        System.err.println("no");
    }
}

public class HappyState extends PartnerState {

    public void whatz4DinDin() {
        System.err.println("chicken");
    }
    public void wannaGo2PDX() {
        System.err.println("yeah!!!");
    }
}

public class SleepState extends PartnerState {

    public void whatz4DinDin() {
        System.err.println("mumble");
    }
    public void wannaGo2PDX() {
        System.err.println("mummph");
    }
}
the State Pattern
the State Pattern

class Partner {

the State Pattern

class Partner {
    private GrumpyState grumpyState;
}
class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
}
the State Pattern

class Partner {
  private GrumpyState grumpyState;
  private HappyState happyState;
  private PartnerState currentState;
}
the State Pattern

class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;
}
the State Pattern

class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
    private PartnerState currentState;

    public Partner() {
}
class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState();
    }
}
the State Pattern

class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState  = new GrumpyState();
        happyState   = new HappyState();
class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState();
        happyState = new HappyState();
        currentState = happyState;
    }
}
the State Pattern

class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState  = new GrumpyState();
        happyState   = new HappyState();
        currentState = happyState;
    }
}
the State Pattern

class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState();
        happyState = new HappyState();
        currentState = happyState;
    }
}
class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState();
        happyState = new HappyState();
        currentState = happyState;
    }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState();
        happyState = new HappyState();
        currentState = happyState;
    }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX() { currentState.wannaGo2PDX(); }
}
the State Pattern

class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState();
        happyState = new HappyState();
        currentState = happyState;
    }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX() { currentState.wannaGo2PDX(); }
}
the State Pattern

class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState  = new GrumpyState();
        happyState   = new HappyState();
        currentState = happyState;
    }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX()  { currentState.wannaGo2PDX(); }

    public boolean negate(boolean v) { return !v; }
}
class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState  = new GrumpyState();
        happyState   = new HappyState();
        currentState = happyState;
    }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX() { currentState.wannaGo2PDX(); }

    public boolean negate(boolean v) { return !v; }
    public int     addOne(int i)     { return i+1; }
}
the State Pattern

class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState();
        happyState = new HappyState();
        currentState = happyState;
    }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX() { currentState.wannaGo2PDX(); }

    public boolean negate(boolean v) { return !v; }
    public int addOne(int i) { return i+1; }
}
the State Pattern

class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState  = new GrumpyState();
        happyState   = new HappyState();
        currentState = happyState;
    }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX()  { currentState.wannaGo2PDX(); }

    public boolean negate(boolean v) { return !v; }
    public int     addOne(int i)     { return i+1; }
}
class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState  = new GrumpyState();
        happyState   = new HappyState();
        currentState = happyState;
    }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX()  { currentState.wannaGo2PDX(); }

    public boolean negate(boolean v) { return !v; }
    public int     addOne(int i)     { return i+1; }
}
the State Pattern

class Partner {
    private GrumpyState  grumpyState;
    private HappyState   happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState  = new GrumpyState();
        happyState   = new HappyState();
        currentState = happyState;
    }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX()  { currentState.wannaGo2PDX(); }

    public boolean negate(boolean v) { return !v; }
    public int     addOne(int i)     { return i+1; }
}

so how to change the state variable currentState?
changing states
changing states

1) the class Partner can change its state on its own
changing states

1) the class Partner can change its state on its own
   currentState = grumpyState;
changing states

1) the class Partner can change its state on its own
currentState = grumpyState;
with only private access to its state variables
changing states

1) the class Partner can change its state on its own
   currentState = grumpyState;
   with only private access to its state variables
changing states

1) the class Partner can change its state on its own
   currentState = grumpyState;
   with only private access to its state variables

2) anyone, including a visitor might change the private
   state variables, provided there are public setter methods
changing states

1) the class Partner can change its state on its own
   currentState = grumpyState;
   with only private access to its state variables

2) anyone, including a visitor might change the private
   state variables, provided there are public setter methods
   makeGrumpy()
changing states

1) the class Partner can change its state on its own
   currentState = grumpyState;
   with only private access to its state variables

2) anyone, including a visitor might change the private
   state variables, provided there are public setter methods
      makeGrumpy()
      makeHappy()
changing states

1) the class Partner can change its state on its own
   currentState = grumpyState;
   with only private access to its state variables

2) anyone, including a visitor might change the private
   state variables, provided there are public setter methods
   makeGrumpy()
   makeHappy()
changing states

1) the class Partner can change its state on its own
   currentState = grumpyState;
   with only private access to its state variables

2) anyone, including a visitor might change the private
   state variables, provided there are public setter methods
      makeGrumpy()
      makeHappy()

3) the Partner’s private states themselves might decide to
   change the parent Partner’s state (behind its back)
changing states

1) the class Partner can change its state on its own
   currentState = grumpyState;
   with only private access to its state variables

2) anyone, including a visitor might change the private
   state variables, provided there are public setter methods
      makeGrumpy()
      makeHappy()

3) the Partner’s private states themselves might decide to
   change the parent Partner’s state (behind its back)
   that requires States to know their parents
changing states

1) the class Partner can change its state on its own
   currentState = grumpyState;
   with only private access to its state variables

2) anyone, including a visitor might change the private
   state variables, provided there are public setter methods
   makeGrumpy()
   makeHappy()

3) the Partner’s private states themselves might decide to
   change the parent Partner’s state (behind its back)
   that requires States to know their parents
privately settable states

class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState();
        happyState = new HappyState();
        currentState = happyState;
    }

    private void setGrumpy() { currentState = grumpyState; }
    private void setHappy() { currentState = happyState; }

    public void whatz4DinDin() {
        currentState.whatz4DinDin();
        setGrumpy();
    }

    public void wannaGo2PDX() {
        currentState.wannaGo2PDX();
        setHappy();
    }
}
publicly settable states

class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState();
        happyState = new HappyState();
        currentState = happyState;
    }

    public void setGrumpy() { currentState = grumpyState; }
    public void setHappy() { currentState = happyState; }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX() { currentState.wannaGo2PDX(); }

    where anybody can change the state at any time ...
states changing their parent's state
states changing their parent's state

abstract public class PartnerState {

states changing their parent’s state

abstract public class PartnerState {
    protected Partner parent;
}
states changing their parent’s state

abstract public class PartnerState {
    protected Partner parent;
states changing their parent’s state

abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
}
states changing their parent’s state

abstract public class PartnerState {
protected Partner parent;

public PartnerState(Partner parent) { this.parent = parent; }
abstract public void whatz4DinDin();
abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
states changing their parent’s state

abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}
states changing their parent’s state

abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}
abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) {
        this.parent = parent;
    }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
states changing their parent's state

abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }  
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); } 
}
states changing their parent’s state

abstract public class PartnerState {
  protected Partner parent;

  public PartnerState(Partner parent) { this.parent = parent; }
  abstract public void whatz4DinDin();
  abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
  public GrumpyState(Partner parent) { super(parent); }
}
states changing their parent’s state

abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }
}
abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }
}
states changing their parent’s state

abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }

    public void wannaGo2PDX() { }
abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }

    public void wannaGo2PDX() {
        System.err.println("no");
    }
abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }

    public void wannaGo2PDX() {
        System.err.println("no");
        parent.setHappy(); // no longer grumpy
abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }

    public void wannaGo2PDX() {
        System.err.println("no");
        parent.setHappy(); // no longer grumpy
    }
}
states changing their parent's state

abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }

    public void wannaGo2PDX() {
        System.err.println("no");
        parent.setHappy(); // no longer grumpy
    }
}
abstract public class PartnerState {
  protected Partner parent;

  public PartnerState(Partner parent) { this.parent = parent; }
  abstract public void whatz4DinDin();
  abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
  public GrumpyState(Partner parent) { super(parent); }

  public void whatz4DinDin() { System.err.println("road kill."); }

  public void wannaGo2PDX() {
    System.err.println("no");
    parent.setHappy(); // no longer grumpy
  }
}
states changing their parent’s state

abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }

    public void wannaGo2PDX() {
        System.err.println("no");
        parent.setHappy(); // no longer grumpy
    }
}

and maybe HappyState can change parent to GrumpyState as well ...
states changing their parent’s state

abstract public class PartnerState {
    protected Partner parent;

    public PartnerState(Partner parent) { this.parent = parent; }
    abstract public void whatz4DinDin();
    abstract public void wannaGo2PDX();
}

public class GrumpyState extends PartnerState {
    public GrumpyState(Partner parent) { super(parent); }

    public void whatz4DinDin() { System.err.println("road kill."); }

    public void wannaGo2PDX() {
        System.err.println("no");
        parent.setHappy(); // no longer grumpy
    }
}

and maybe HappyState can change parent to GrumpyState as well ...
states changing their parent’s state

class Partner {
    private GrumpyState grumpyState;
    private HappyState happyState;
    private PartnerState currentState;

    public Partner() {
        grumpyState = new GrumpyState(this);
        happyState = new HappyState(this);
        currentState = happyState;
    }

    public void setGrumpy() { currentState = grumpyState; }
    public void setHappy() { currentState = happyState; }

    public void whatz4DinDin() { currentState.whatz4DinDin(); }
    public void wannaGo2PDX() { currentState.wannaGo2PDX(); }
}

where the states change the Partner’s state without it knowing!
a soft drink dispenser
a soft drink dispenser

// CMDriver.java
a soft drink dispenser

// CMDriver.java
a soft drink dispenser

// CMDriver.java

package cm;
a soft drink dispenser

// CMDriver.java

package cm;
a soft drink dispenser

// CMDriver.java

package cm;

public class CMDriver {

a soft drink dispenser

// CMDriver.java

package cm;

public class CMDriver {
    public static void main (String args[]) {

a soft drink dispenser

// CMDriver.java

package cm;

class CMDriver {

    public static void main (String args[]) {

}
a soft drink dispenser

// CMDriver.java

package cm;

public class CMDriver {
    public static void main (String args[]) {
        Dispenser d = new Dispenser();
    }
}
a soft drink dispenser

// CMDriver.java

package cm;

public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();
    }
}
a soft drink dispenser

// CMDriver.java
package cm;

class CMDriver {
    public static void main (String args[]) {
        Dispenser d = new Dispenser();
        d.insertCoin(new Quarter());
    }
}
package cm;

public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
    }
}
a soft drink dispenser

// CMDriver.java

package cm;

public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
    }
}

a soft drink dispenser

// CMDriver.java

package cm;

public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.pushRefund();
    }
package cm;

public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.pushRefund();
    }
}

a soft drink dispenser
package cm;

public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.pushRefund();

        d.insertCoin(new Quarter());
    }
}
public class CMDriver {
    public static void main (String args[]) {
        Dispenser d = new Dispenser();
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.pushRefund();
        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
    }
}
package cm;

public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.pushRefund();

        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
    }
}
a soft drink dispenser

// CMDriver.java

class CMDriver {
    public static void main(String[] args) {
        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.pushRefund();

        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
        d.pushSelection(1);
    }
}
a soft drink dispenser

// CMDriver.java

dispenser cm;

public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.pushRefund();

        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
        d.pushSelection(1);
    }
}
a soft drink dispenser

// CMDriver.java

package cm;

public class CMDriver {
    public static void main (String args[]) {

        Dispenser d = new Dispenser();

        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.insertCoin(new Quarter());
        d.pushSelection(1);
        d.pushRefund();

        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
        d.insertCoin(new Quarter());
        d.pushSelection(1);
    }
}

public static void main(String args[]) {

    Dispenser d = new Dispenser();
    d.insertCoin(new Quarter());
    d.pushSelection(1);
    d.insertCoin(new Quarter());
    d.pushSelection(1);
    d.pushRefund();

    d.insertCoin(new Quarter());
    d.insertCoin(new Quarter());
    d.insertCoin(new Quarter());
    d.insertCoin(new Quarter());
    d.pushSelection(1);
}

a soft drink dispenser
a soft drink dispenser
a soft drink dispenser

java cm/CMDriver
a soft drink dispenser

java cm/CMDriver
coin inserted...
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
a soft drink dispenser

```
java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
```
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
Refunding coins...
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
Refunding coins...
coin inserted...
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
Refunding coins...
coin inserted...
coin inserted...
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
Refunding coins...
coin inserted...
coin inserted...
coin inserted...
coin inserted...
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
Refunding coins...
coin inserted...
coin inserted...
coin inserted...
coin inserted...
Please make a selection...

Friday, May 14, 2010
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
Refunding coins...
coin inserted...
coin inserted...
coin inserted...
coin inserted...
Please make a selection...
Selection pushed...
a soft drink dispenser

java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
Refunding coins...
coin inserted...
coin inserted...
coin inserted...
Please make a selection...
Selection pushed...
Dispensing item...
java cm/CMDriver
coin inserted...
Selection pushed...
Please insert more coins
coin inserted...
Selection pushed...
Please insert more coins
Refund pushed...
Refunding coins...
coin inserted...
coin inserted...
coin inserted...
coin inserted...
Please make a selection...
Selection pushed...
Dispensing item...
a soft drink dispenser
a soft drink dispenser

package cm;
a soft drink dispenser

package cm;
a soft drink dispenser

package cm;

abstract public class Coin {

a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;
}
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;
}
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
}
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
    public Coin(int value) { this.value = value; }
}
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
    public Coin(int value) { this.value = value; }
}
package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
    public Coin(int value) { this.value = value; }

    public int getValue() { return value; }
}
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
    public Coin(int value) { this.value = value; }

    public int getValue() { return value; }
}

Friday, May 14, 2010
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
    public Coin(int value) { this.value = value; }

    public int getValue() { return value; }
}

Friday, May 14, 2010
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
    public Coin(int value) { this.value = value; }

    public int getValue() { return value; }
}

Friday, May 14, 2010
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
    public Coin(int value) { this.value = value; }

    public int getValue() { return value; }
}

public class Quarter extends Coin {

a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {}
    public Coin(int value) { this.value = value; }

    public int getValue() { return value; }
}

public class Quarter extends Coin {
    public Quarter() { super(25); }
}
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {};
    public Coin(int value) { this.value = value; }

    public int getValue() { return value; }
}

public class Quarter extends Coin {
    public Quarter() { super(25); }
}
a soft drink dispenser

package cm;

abstract public class Coin {
    private int value;

    public Coin() {
    }
    public Coin(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

public class Quarter extends Coin {
    public Quarter() {
        super(25);
    }
}
a soft drink dispenser
a soft drink dispenser

public void transferToSafe() {

a soft drink dispenser

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
a soft drink dispenser

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        // code
    }
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
    }
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
    }
}
a soft drink dispenser

```java
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
    }
}
```
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
}
a soft drink dispenser

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}
a soft drink dispenser

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}
a soft drink dispenser

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        totalValue += c.getValue();
        safe.add(c);
        it.remove();
    }
    System.out.println("safe now has "+totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    }
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has "+totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
    }
}
a soft drink dispenser

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has "+ totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding "+ c.getValue());
        it.remove();
    }
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
    System.out.println("Coin box refund completed " + tempValue + " left");
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
    System.out.println("Coin box refund completed " + tempValue + " left");
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has "+totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
    System.out.println("Coin box refund completed " + tempValue + " left");
}
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

class Coin {
    public int getValue() {
        return tempValue;
    }
}

class Safe {
    public void addCoin(Coin c) {
        totalValue += c.getValue();
        safe.add(c);
    }
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
    System.out.println("Coin box refund completed " + tempValue + " left");
}

public int getValue() { return tempValue; }
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
    System.out.println("Coin box refund completed " + tempValue + " left");
}

public int getValue() { return tempValue; }
public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
    System.out.println("Coin box refund completed " + tempValue + " left");
}

public int getValue() { return tempValue; }
a soft drink dispenser

public void transferToSafe() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        it.remove();
        totalValue += c.getValue();
        safe.add(c);
    }
    System.out.println("safe now has " + totalValue);
}

public void refund() {
    Iterator<Coin> it = temp.iterator();
    while (it.hasNext()) {
        Coin c = it.next();
        System.out.println("refunding " + c.getValue());
        it.remove();
        tempValue -= c.getValue();
    }
    System.out.println("Coin box refund completed " + tempValue + " left");
}

public int getValue() { return tempValue; }
public interface DispenserI { 
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State currentState;
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State currentState;
    private Ready ready;

public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State currentState;
    private Ready ready;
    private Accepting accepting;
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;
}
Dispenser.java

public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;

    public Dispenser() {

Dispenser.java

public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;

    public Dispenser() {
        coinBox   = new CoinBox();
    }
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;

    public Dispenser() {
        coinBox   = new CoinBox();
        ready     = new Ready(this);
    }
}

Friday, May 14, 2010
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

global class Dispenser implements DispenserI {
    private State currentState;
    private Ready ready;
    private Accepting accepting;
    private Selecting selecting;
    private CoinBox coinBox;

    public Dispenser() {
        coinBox = new CoinBox();
        ready = new Ready(this);
        accepting = new Accepting(this);
    }
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;

    public Dispenser() {
        coinBox   = new CoinBox();
        ready     = new Ready(this);
        accepting = new Accepting(this);
        selecting = new Selecting(this);
    }
}
Dispenser.java

public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;

    public Dispenser() {
        coinBox   = new CoinBox();
        ready     = new Ready(this);
        accepting = new Accepting(this);
        selecting = new Selecting(this);
        setReady();
    }
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State currentState;
    private Ready ready;
    private Accepting accepting;
    private Selecting selecting;
    private CoinBox coinBox;

    public Dispenser() {
        coinBox = new CoinBox();
        ready = new Ready(this);
        accepting = new Accepting(this);
        selecting = new Selecting(this);
        setReady();
    }
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;

    public Dispenser() {
        coinBox   = new CoinBox();
        ready     = new Ready(this);
        accepting = new Accepting(this);
        selecting = new Selecting(this);
        setReady();
    }
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;

    public Dispenser() {
        coinBox   = new CoinBox();
        ready     = new Ready(this);
        accepting = new Accepting(this);
        selecting = new Selecting(this);
        setReady();
    }

    public void insertCoin(Coin c) {

public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State    currentState;
    private Ready    ready;
    private Accepting accepting;
    private Selecting selecting;
    private CoinBox  coinBox;

    public Dispenser() {
        coinBox    = new CoinBox();
        ready      = new Ready(this);
        accepting  = new Accepting(this);
        selecting  = new Selecting(this);
        setReady();
    }

    public void insertCoin(Coin c) {
        System.out.println("coin inserted...");
    }
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting accepting;
    private Selecting selecting;
    private CoinBox    coinBox;

    public Dispenser() {
        coinBox   = new CoinBox();
        ready     = new Ready(this);
        accepting = new Accepting(this);
        selecting = new Selecting(this);
        setReady();
    }

    public void insertCoin(Coin c) {
        System.out.println("coin inserted...");
        currentState.accept(c);
    }
}
public interface DispenserI {
    final static int BEVERAGE_PRICE = 75;
}

public class Dispenser implements DispenserI {
    private State      currentState;
    private Ready      ready;
    private Accepting  accepting;
    private Selecting  selecting;
    private CoinBox    coinBox;

    public Dispenser() {
        coinBox   = new CoinBox();
        ready     = new Ready(this);
        accepting = new Accepting(this);
        selecting = new Selecting(this);
        setReady();
    }

    public void insertCoin(Coin c) {
        System.out.println("coin inserted...");
        currentState.accept(c);
    }
}
public void pushSelection(int selection) {
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
}
Dispenser.java

public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
Dispenser.java

public void pushSelection(int selection) {
    System.out.println("Selection pushed... ");
    currentState.select(selection);
}
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
Dispenser.java

public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}
public void pushSelection(int selection) {
    System.out.println("Selection pushed... ");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}
public class Dispenser {
    private State currentState;

    public Dispenser(State initial) {
        currentState = initial;
    }

    public void pushSelection(int selection) {
        System.out.println("Selection pushed...");
        currentState.select(selection);
    }

    public void pushRefund() {
        System.out.println("Refund pushed...");
        currentState.refund();
    }

    public boolean readyToDispense() {
        return true;
    }
}

Friday, May 14, 2010
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}

public boolean dispense(int selection) {
Dispenser.java

public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}

public boolean dispense(int selection) {
    System.out.println("Dispensing item...");
}
Dispenser.java

public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}

public boolean dispense(int selection) {
    System.out.println("Dispensing item...");
    // do mechanical dispensing here
Dispenser.java

public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}

public boolean dispense(int selection) {
    System.out.println("Dispensing item...");
    // do mechanical dispensing here
    coinBox.transferToSafe();
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}

public boolean dispense(int selection) {
    System.out.println("Dispensing item...");
    // do mechanical dispensing here
    coinBox.transferToSafe();
    return true;
}
public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}

public boolean dispense(int selection) {
    System.out.println("Dispensing item...");
    // do mechanical dispensing here
    coinBox.transferToSafe();
    return true;
}
Dispenser.java

public void pushSelection(int selection) {
    System.out.println("Selection pushed...");
    currentState.select(selection);
}

public void pushRefund() {
    System.out.println("Refund pushed...");
    currentState.refund();
}

public boolean readyToDispense() {
    return coinBox.getValue() >= BEVERAGE_PRICE;
}

public boolean dispense(int selection) {
    System.out.println("Dispensing item...");
    // do mechanical dispensing here
    coinBox.transferToSafe();
    return true;
}
public void refund() {

public void refund() {
    coinBox.refund();
public void refund() {
    coinBox.refund();
}
public void refund() {
    coinBox.refund();
}

dispenser.java

public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }

void setState(State s) { currentState = s; }
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }

void setState(State s) { currentState = s; }
Dispenser.java

public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }

void setState(State s) { currentState = s; }

void setReady() { currentState = ready; }

Friday, May 14, 2010
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) {
    coinBox.add(c);
}

void setState(State s) {
    currentState = s;
}

void setReady() {
    currentState = ready;
}

void setAccepting() {
    currentState = accepting;
}
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }

void setState(State s) { currentState = s; }

void setReady() { currentState = ready; }
void setAccepting() { currentState = accepting; }
void setSelecting() { currentState = selecting; }
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }

void setState(State s) { currentState = s; }
void setReady() { currentState = ready; }
void setAccepting() { currentState = accepting; }
void setSelecting() { currentState = selecting; }
}
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }

void setState(State s) { currentState = s; }
void setReady() { currentState = ready; }
void setAccepting() { currentState = accepting; }
void setSelecting() { currentState = selecting; }
}
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) { coinBox.add(c); }
public void refund() {
    coinBox.refund();
}

public void accept(Coin c) {
    coinBox.add(c);
}

void setState(State s) {
    currentState = s;
}

void setReady() {
    currentState = ready;
}
void setAccepting() {
    currentState = accepting;
}
void setSelecting() {
    currentState = selecting;
}
the abstract State class
the abstract State class

package cm;
the abstract State class

package cm;
the abstract State class

package cm;

abstract public class State {

the abstract State class

package cm;

abstract public class State {
    protected Dispenser owner;
package cm;

abstract public class State {
    protected Dispenser owner;
}
package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
}
package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }
}
the abstract State class

package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }
}
package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }
}
the abstract State class

package cm;

abstract public class State {
   protected Dispenser owner;

   public State(Dispenser d) {
      owner = d;
   }

   abstract public void accept(Coin c);
}
the abstract State class

dumped:

```java
package cm;

abstract public class State {
  protected Dispenser owner;

  public State(Dispenser d) {
    owner = d;
  }

  abstract public void accept(Coin c);
  abstract public void select(int n);
```

Friday, May 14, 2010
package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);
package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);

    public void refund() {
    }
the abstract State class

package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);

    public void refund() {
        System.out.println("Refunding coins...");
    }
}
the abstract State class

package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);

    public void refund() {
        System.out.println("Refunding coins...");
        owner.refund();
    }
}
package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);

    public void refund() {
        System.out.println("Refunding coins...");
        owner.refund();
        owner.setReady();
    }
}
the abstract State class

package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);

    public void refund() {
        System.out.println("Refunding coins...");
        owner.refund();
        owner.setReady();
    }
}
the abstract State class

package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);

    public void refund() {
        System.out.println("Refunding coins...");
        owner.refund();
        owner.setReady();
    }
}
the abstract State class

package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);

    public void refund() {
        System.out.println("Refunding coins...");
        owner.refund();
        owner.setReady();
    }
}

the abstract State class

package cm;

abstract public class State {
    protected Dispenser owner;

    public State(Dispenser d) {
        owner = d;
    }

    abstract public void accept(Coin c);
    abstract public void select(int n);

    public void refund() {
        System.out.println("Refunding coins...");
        owner.refund();
        owner.setReady();
    }
}
Ready and Accepting states
Ready and Accepting states

public class Ready extends State {

public class Ready extends State {


Ready and Accepting states

public class Ready extends State {

    public Ready(Dispenser d) { super(d); }
}
Ready and Accepting states

public class Ready extends State {
    public Ready(Dispenser d) { super(d); }
}
public class Ready extends State {
    public Ready(Dispenser d) { super(d); }
    public void accept(Coin c) {
    }
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
    }
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }
}
public class Ready extends State {

public Ready(Dispenser d) { super(d); }

public void accept(Coin c) {
    owner.accept(c);
    owner.setAccepting();
}

Ready and Accepting states

public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
Ready and Accepting states

public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}
Ready and Accepting states

public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}

public class Accepting extends State {

public class Ready extends State {
    public Ready(Dispenser d) { super(d); }
    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }
    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
Ready and Accepting states

```java
public class Ready extends State {
    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
    }
}
```
public class Ready extends State {
    public Ready(Dispenser d) { super(d); }
    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }
    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }
    public void accept(Coin c) {
        owner.accept(c);
        if (owner.readyToDispense()) {

        }
    }
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        if (owner.readyToDispense()) {
            System.out.println("Please make a selection...");
        }
    }
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        if (owner.readyToDispense()) {
            System.out.println("Please make a selection...".toArray);
            owner.setSelecting();
        }
    }
}
Ready and Accepting states

public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        if (owner.readyToDispense()) {
            System.out.println("Please make a selection...");
            owner.setSelecting();
        }
    }
}
Ready and Accepting states

public class Ready extends State {
    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}  
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        if (owner.readyToDispense()) {
            System.out.println("Please make a selection...");
            owner.setSelecting();
        }
    }
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        if (owner.readyToDispense()) {
            System.out.println("Please make a selection...");
            owner.setSelecting();
        }
    }
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {
    }
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        if (owner.readyToDispense()) {
            System.out.println("Please make a selection...");
            owner.setSelecting();
        }
    }

    public void select(int n) { System.out.println("Please insert more coins"); }
}
public class Ready extends State {

    public Ready(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        owner.setAccepting();
    }

    public void select(int n) {}
}

public class Accepting extends State {
    public Accepting(Dispenser d) { super(d); }

    public void accept(Coin c) {
        owner.accept(c);
        if (owner.readyToDispense()) {
            System.out.println("Please make a selection...");
            owner.setSelecting();
        }
    }

    public void select(int n) { System.out.println("Please insert more coins"); }
}
Selecting state
Selecting state
Selecting state

package cm;
Selecting state

package cm;
package cm;

public class Selecting extends State {
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) {
        super(d);
    }
}
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }
}
Selecting state

package cm;

class Selecting extends State {
    public Selecting(Dispenser d) {
        super(d);
    }

    // in the selecting state, you can put in extra money,
Selecting state

package cm;

public class Selecting extends State {
   public Selecting(Dispenser d) { super(d); }

   // in the selecting state, you can put in extra money,
   // but it only has to be refunded later
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) {
        super(d);
    }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {

package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }
}
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }
}
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) {
        super(d);
    }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }
}
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
Selecting state

package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) {
        super(d);
    }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) {
        super(d);
    }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
    public void select(int n) {

Selecting state

package cm;

class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
    public void select(int n) {
        if (owner.readyToDispense()) {

    }
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }

    // in the selecting state, you can put in extra money, 
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
    public void select(int n) {
        if (owner.readyToDispense()) {
            owner.dispense(n);
        }
    }
}
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
    public void select(int n) {
        if (owner.readyToDispense()) {
            owner.dispense(n);
            owner.setReady();
        }
    }
}
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) {
        super(d);
    }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
    public void select(int n) {
        if (owner.readyToDispense()) {
            owner.dispense(n);
            owner.setReady();
        }
    }
}
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) {
        super(d);
    }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
    public void select(int n) {
        if (owner.readyToDispense()) {
            owner.dispense(n);
            owner.setReady();
        }
    }
}
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) {
        super(d);
    }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
    public void select(int n) {
        if (owner.readyToDispense()) {
            owner.dispense(n);
            owner.setReady();
        }
    }
}
Selecting state

package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }

    // in the selecting state, you can put in extra money,
    // but it only has to be refunded later
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money
    // if so, dispenses and resets to ready state
    public void select(int n) {
        if (owner.readyToDispense()) {
            owner.dispense(n);
            owner.setReady();
        }
    }
}

Friday, May 14, 2010
package cm;

public class Selecting extends State {
    public Selecting(Dispenser d) { super(d); }

    // in the selecting state, you can put in extra money, 
    // but it only has to be refunded later 
    public void accept(Coin c) {
        owner.accept(c);
    }

    // if you try to select, it checks if enough money 
    // if so, dispenses and resets to ready state 
    public void select(int n) {
        if (owner.readyToDispense()) {
            owner.dispense(n);
            owner.setReady();
        }
    }
}
responsibility for changing state

from the text (p. 412):

“The disadvantage of having state transitions in the state classes is that we create dependencies between the state classes.”