have one single new class, AlternatingRateStrategy, that I must question the reliability of. Its complexity is low and I can do semi-automatic testing of it so I am confident. But I have admittedly no fully automated testing of it. I must return later to solve this reliability issue.

- *Maintainability.* The implementation is simple and straight forward—except perhaps for the calendar checking code, but this code is required in all solutions. It is much easier to maintain code that you understand.
- *Client's interface consistent.* A very important aspect of this proposal is that the pay station's use of rate calculation objects is unchanged. It has stayed the same even in the face of a new complex requirement. Complex and simple rate calculations are treated uniformly.
- *Reuse.* I have once and for all written the strategy to handle weekday/weekend variations. A new town that is interested in this behavior but with other concrete rate calculations for the respective two periods of time is a simple matter of configuration in the constructor call.
- *Flexible and open for extension.* The present solution is open for new requirements by *Change by addition.* This is in contrast to a parametric solution where the pay station's production code would have to be modified to support new states.
- *State specific behavior is localized.* All the behavior that is depending on the state of the system clock is encapsulated in the AlternatingRateStrategy. As the only aspect of the Gammatown requirements that relates to system time is indeed the rate calculation it is the right place. To put it in other words, the cohesion is high.

Some of the liabilities:

- *Increased number of objects.* This is the standard problem with compositional design namely that it introduces more classes and objects. Knowing your patterns, as described in Chapter 18, is a way to reduce this problem.

11.9 The State Pattern

In the discussion above I once again used the ③-①-② process as I did when I discovered the STRATEGY pattern. This time it was in particular the ② part that I elaborated upon: *thinking collaboration.* Instead of the AlternatingRateStrategy doing all the calculations itself it *delegated* the concrete rate calculations to delegate objects and itself concentrated on the task of deciding which delegate to use depending on the clock. This perspective on AlternatingRateStrategy is focused on how it fulfills its responsibility but let me turn the table and instead look at what it appears to do seen from the pay station's perspective. In essence, the pay station sees an *object that behaves differently depending on its internal state*: is it in its weekend state or is it in its weekday state? This formulation is exactly the intent of the STATE pattern: *Allow an object to alter its behavior when its internal state changes.* The STATE pattern is summarized in design pattern box 11.1 on page 25.

The state pattern defines two central roles: the **context** and the **state**. The context object delegate requests to its current state object, and internal state changes are affected by changing the concrete state object. In my pay station case, the AlternatingRateStrategy is the context while LinearRateStrategy and ProgressiveRateStrategy are state objects. It should be noted that the structure of the STATE pattern (see the pattern box on page 25) is more general than the structure in the pay station: in the pay station, the context object also implements the state interface but this is not always the case. Also the new state is "calculated" each time the calculateTime method is invoked which is also not a requirement of the pattern.

An important observation is that the STATE pattern structurally is equivalent to the STRATEGY pattern! Look at the UML class diagrams for the two patterns: they are alike. This is not surprising as it is the same compositional design idea, the ③-①-② process, that leads to both. So, the interesting question is why do we speak of two different patterns? The answer is that it is two different *problems* that this compositional structure is a solution to. STRATEGY's intent is to handle *variability of business rules or algorithms* whereas STATE's intent is to provide *behavior that varies according to object's internal state*. The rate policy is a variation in business rules used by Alphatown and Betatown, hence the STRATEGY pattern. The problem of Gammatown in contrast is to pick the proper behavior based upon the pay stations internal state, the clock, hence the STATE pattern.

> **Key Point: Design patterns are defined by the problems they solve**
>
> *It is the characteristics of the problem a design pattern aims to solve, its intent, that define the proper pattern to apply.*

11.10 State Machines

The STATE pattern is a compositional way of implementing **state machines**. State machines are seen in many real life and computational contexts and describe systems that change between different states as a response to external events. A classic and simple case is a subway turnstile that grants access to a subway station only when a traveler inserts a coin. A transition table shows how the turnstile reacts and changes state:

Current State	Event	New State	Action
Locked	Coin entered	Unlocked	Unlock arms
Unlocked	Person passes	Locked	Lock arms
Locked	Person passes	Locked	Sound alarm
Unlocked	Coin entered	Unlocked	Return coin

That is, the turnstile can be in one of two states: "Locked" and "Unlocked" and it changes state depending on the events defined in the table, like entering a coin or passing the arms. The state change itself has an associated action, like sounding the alarm, unlocking the turnstile arms, etc. The STATE pattern can be used to implement the state machine behavior. The turnstile class simply forwards the "coin" and "pass" events to its state object, while the locked and unlocked state objects take appropriate action as well as ensure state changes. An example implementation is shown in Figure 11.4 on page 26.

The state pattern does not dictate which object is responsible for making the state change: the context object or the concrete-state objects. In my pay station case, the only right place to put the state changing code is in the context object, AlternatingRateStrategy: putting it in the concrete state objects, LinearRateStrategy and ProgressiveRateStrategy, would make them incohesive and make them malfunction in a Alphatown/Betatown setting. However, in many state machines it is the individual concrete-state object that knows the proper next state. In this case the concrete-state objects must of course have a reference to the context object in order to tell it to change state.

11.11 Summary of Key Concepts

It is a recurring situation that a requirement for behavior is a combination of behavior already developed. In such a case, reusing the existing code becomes important. In this chapter I have analyzed the polymorphic proposal and it turns out that it is very difficult to achieve a satisfactory design. The polymorphic variants suffer from either code duplication or low cohesion. The code duplication is partly due to the lack of multiple implementation inheritance in modern object-oriented languages like Java and C#; however even the multiple inheritance solution suffers from low maintainability. One of the low cohesion solutions often seen in practice leads to a design where behavior and methods that is actually unique to the subclasses nevertheless bubbles up into an ever growing (abstract) superclass that therefore suffers low cohesion.

The compositional proposal, based upon the ③-①-② process, provides a cleaner way by suggesting to *compose complex behavior from simpler behaviors.* An often useful metaphor is to consider software design as a company or organization consisting of coordinators and specialists that collaborate to get the job done. In the same vein the corner stone of compositional design is *objects that collaborate*.

In the concrete pay station case, the resulting design is an application of the STATE pattern. The STATE pattern describes a solution to the problem of *making an object change behavior according to its internal state.* In the pay station case, the rate calculation changes behavior according to the state of the system clock. The STATE pattern defines two roles, the **context** and the **state** role and require the context object to delegate all state dependent requests to the current state object. The context changes state by changing the object implementing the state role and thus appears to change behavior. Often it is the context itself that changes the state object reference, but it can also be some of the concrete state objects that do it. The STATE pattern is often used to implement *state machines.*

11.12 Selected Solutions

Discussion of Exercise 11.2:

The problem with this solution is that if we want to be consistent in the way we handle new rate requirements (and consistency is the way to keep your code understandable) then the abstract superclass just grows bigger and bigger as it fills up with

rate calculation methods. You get *method bloat* in the superclass. This is also a tendency that is often seen in practice. The result is a class with lower cohesion as it contains methods unrelated to itself.

A superclass bloated with methods that are not relevant for itself but only for a large set of subclasses is less understandable. And—new rate requirements will lead to *change by modification* in the superclass.

There is also a risk that it becomes a junk pile of dead code. Consider that "FortyThreeTown" no longer uses our pay station product. Maybe I remember to remove the pay station subclass that was running there, but do I also remember to remove the methods in the superclass that are no longer used in any product?

Discussion of Exercise 11.3:

In the current proposal, it is the time of the last entered coin that dictates the rate policy used. So if the driver starts entering coins on Friday but ends on a Saturday it is the weekend rate policy that is used for the full amount.

11.13 Review Questions

Describe the problems associated with testing the Gammatown requirement of rate calculation based upon the day of week.

Outline the benefits and liabilities of a polymorphic proposal to handle Gammatown's requirement.

Outline the benefits and liabilities of a proposal that uses a conditional statement in the pay station to determine which rate strategy object to use.

Outline the fully compositional proposal: What are the steps in the ③-①-② process and what is the resulting design? Outline benefits and liabilities of this proposal.

What is the STATE pattern? What problem does it address and what solution does it suggest? What are the roles involved? What are the benefits and liabilities?

11.14 Further Exercises

Exercise 11.4:

Compare the STRATEGY and STATE pattern and identify similarities and differences between them. You should include aspects like the structure (interfaces, classes, and relations), intent, roles, and cost-benefits.

Exercise 11.5. Source code directory:
`exercise/state/alarm`

A simple digital alarm clock has a display, showing "hour:minute", and three buttons marked "mode", "+" and "–". Normally the display shows the time. By pressing "mode" the display instead shows the alarm time and allows setting the alarm hour

by pressing "+" or "–" to increase or decrease the hour. Pressing "mode" a second time allows changing the minutes, and pressing a third time returns the display to showing the time. That is, the alarm clock can be in one of three states: "display time", "set alarm hour", and "set alarm minute" state.

The hardware buttons of the clock invoke methods in the AlarmClock interface:

Listing: exercise/state/alarm/AlarmClock.java

```
/** Interface for a simple alarm clock.
*/
public interface AlarmClock {
  /** return the contents of the display depending on the
   * state of the alarm clock.
   * @return the display contents
   */
  public String readDisplay();

  /** press the "mode" button on the clock */
  public void mode();

  /** press the "increase" (+) button on the clock */
  public void increase();

  /** press the "decrease" (-) button on the clock */
  public void decrease();
}
```

To demonstrate how the interface works, consider the following (learning) test where the present time is "11:32" and the alarm set to "06:15".

```
  @Test
  public void shouldHandleAll() {
    // show time mode
    assertEquals( "11:32", clock.readDisplay() );
    // + and - has no effect in show time mode
    clock.increase();
    assertEquals( "11:32", clock.readDisplay() );
    clock.decrease();
    assertEquals( "11:32", clock.readDisplay() );
    // switch to set hour mode
    clock.mode();
    assertEquals( "06:15", clock.readDisplay() );
    clock.increase(); // increment hour by one
    clock.increase();
    clock.increase();
    assertEquals( "09:15", clock.readDisplay() );
    // switch to set minute mode
    clock.mode();
    clock.decrease();
    clock.decrease();
    clock.decrease();
    clock.decrease();
    clock.decrease();
    assertEquals( "09:10", clock.readDisplay() );
```

```
    // go back to show time mode
    clock.mode();
    assertEquals( "11:32", clock.readDisplay() );
    // remembers the set alarm time.
    clock.mode();
    assertEquals( "09:10", clock.readDisplay() );
  }
```

1. Sketch a STATE pattern based design for implementing the state machine.
2. Implement the production code. You may "fake" the time (like the clock always being "11:32") but the alarm time must be setable.

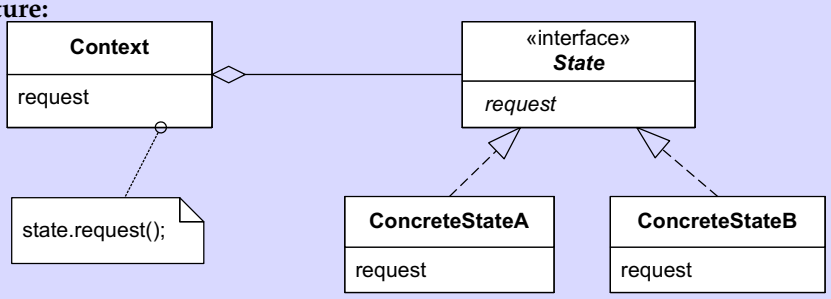
[11.1] Design Pattern: State

Intent Allow an object to alter its behavior when its internal state changes.

Problem Your product's behavior varies at run-time depending upon some internal state.

Solution Describe the responsibilities of the dynamically varying behavior in an interface and implement the concrete behavior associated with each unique state in an object, the state object, that implements this interface. The context object delegates to its current state object. When internal state changes occur, the current state object reference is changed to refer to the corresponding state object.

Structure:



Roles **State** specifies the responsibilities and interface of the varying behavior associated with a state, and **ConcreteState** objects define the specific behavior associated with each specific state. The **Context** object delegates to its current state object. The state object reference is changed whenever the context changes its internal state.

Cost - Benefit *State specific behavior is localized* as all behavior associated with a specific state is in a single class. It *makes state transitions explicit* as assigning the current state object is the only way to change state. A liability is the *increased number of objects and interactions* compared to a state machine based upon conditional statements in the context object.

Listing: chapter/state/turnstile/TurnstileImpl.java

```
/** State pattern implementation of a subway turnstile.
*/
public class TurnstileImpl implements Turnstile {
  State
    lockedState = new LockedState(this),
    unlockedState = new UnlockedState(this),
    state = lockedState;
  public void coin() { state.coin(); }
  public void pass() { state.pass(); }

  public static void main(String[] args) {
    System.out.println( "Demo of turnstile state pattern" );
    Turnstile turnstile = new TurnstileImpl();
    turnstile.coin();
    turnstile.pass();
    turnstile.pass();
    turnstile.coin();
    turnstile.coin();
  }
}

abstract class State implements Turnstile {
  protected TurnstileImpl turnstile;
  public State(TurnstileImpl ts) { turnstile = ts; }
}

class LockedState extends State {
  public LockedState(TurnstileImpl ts) { super(ts); }
  public void coin() {
    System.out.println( "Locked state: Coin accepted");
    turnstile.state = turnstile.unlockedState;
  }
  public void pass() {
    System.out.println( "Locked state: Passenger pass: SOUND ALARM");
  }
}
class UnlockedState extends State {
  public UnlockedState(TurnstileImpl ts) { super(ts); }
  public void coin() {
    System.out.println( "Unlocked state: Coin entered: RETURN IT");
  }
  public void pass() {
    System.out.println( "Unlocked state: Passenger pass");
    turnstile.state = turnstile.lockedState;
  }
}
```

Figure 11.4: Example implementation of turnstile state pattern.