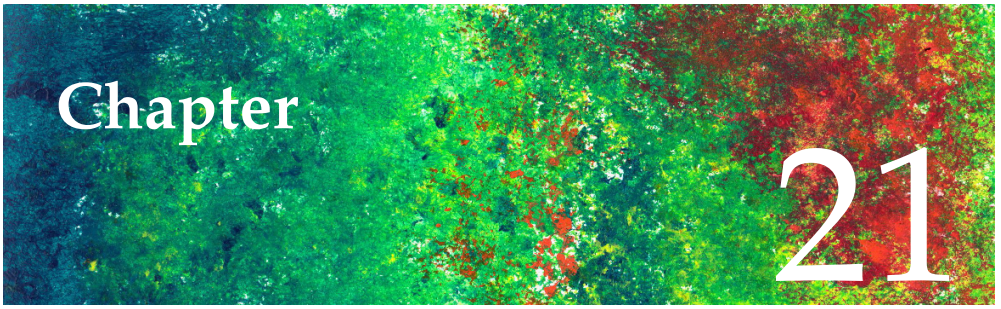


Adapter

21.1 The Problem

Our pay station company has been contacted by Lunatown that wants to buy our pay stations but they have a very peculiar requirement: the rate should be correlated to the phases of the moon. At full moon the rates should equal that of Alphatown while at new moon they should be doubled. In the period in between, the rate should proportionally vary between these extremes. Lunatown has provided us with a Java class for doing the calculation—the problem is that the interface of the class does not follow the conventions used in our own production code:

Fragment: chapter/adapter/src/paystation/thirdparty/LunaRateCalculator.java

```
public int calculateRateForAmount( double dollaramount ) {
```

The parameter, amount, is a double and in dollars, not cents, and the method has a different name. Moreover, the class is not open source, and the municipality will not give us access to the source code as it belongs to a third-party company. However, it is very tempting to use the package instead of trying to figure out the astronomical calculations ourselves.

21.2 Composing a Solution

The pay station design is already well prepared for configuring a product for Lunatown as the rate strategy has already been factored out in our design. However, I cannot simply configure the pay station with an instance of the LunaRateCalculator as it does not implement the RateStrategy interface (it would be pretty odd if it did: it is developed by another company!) nor use the same parameters. What I need to do is to *favor object composition* one step deeper. I put an intermediate object in between to handle the translation process back and forth between the pay station and the luna rate calculator. The concrete adaptation code is pretty simple.

Listing: chapter/adapter/src/paystation/domain/LunaAdapter.java

```
package paystation.domain;
import paystation.thirdparty.*;

/** An adapter for adapting the Lunatown rate calculator
*/
public class LunaAdapter implements RateStrategy {
  private LunaRateCalculator calculator;
  public LunaAdapter() {
    calculator = new LunaRateCalculator();
  }

  public int calculateTime( int amount ) {
    double dollar = amount / 100.0;
    return calculator.calculateRateForAmount( dollar );
  }
}
```

This intermediate object is the **adapter** which is the central role in the ADAPTER pattern.

Exercise 21.1: Draw the UML class diagram and the sequence diagram for the Lunatown adaptation.

21.3 The Adapter Pattern

Adapters are well known for anyone who has struggled with the power plugs around the globe—even in Europe there are many different types of outlets that require special plugs. The software world is much less standardized so in practical software development, adapters play an important role to "glue" software units from different vendors, open source contributors, or even different teams within the same organization, together.

The ADAPTER pattern (design pattern box 21.1, page 23) is a classic example of the *favor object composition* principle. Its intent is

> *Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.*

The adapter contains a reference to the third party object, the **adaptee**, and performs parameter, protocol, and return value translations. The amount of translation made range from simple, such as simple interface conversion (changed method names, different parameters), to complex (different protocols). To give an example of the latter, consider a client that interfaces a sensor, and expects a protocol in which the sensor will deliver measured values every 10 seconds (for instance using an OBSERVER pattern). That is, the client expects active sensor software. If a sensor comes with a software unit that is passive, ie. expects the client to request the measurements, then the adapter becomes much more complex, involving a thread to make the passive adaptee appear active to the client.

A given adapter can usually adapt all subclasses of the adaptee, and is thus reusable. Of course, it cannot be reused for other adaptee classes.

For the sake of completeness, the ADAPTER pattern presented here is what is called an *object adapter* by Gamma et al. They also discuss a *class adapter* that relies on multiple inheritance but this is of course not possible in Java nor C#.

21.4 Selected Solutions

Discussion of Exercise 21.1:

The class diagram is shown in Figure 21.1.

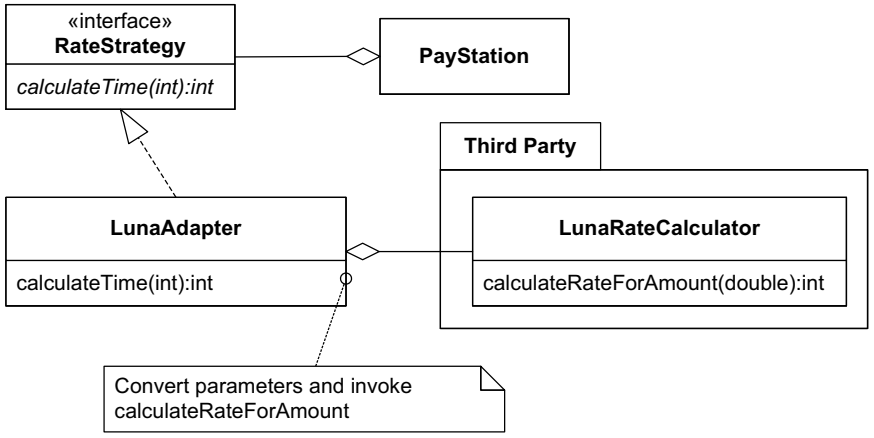


Figure 21.1: Adapting the Lunatown rate calculator.

21.5 Review Questions

Describe the ADAPTER pattern. What problem does it solve? What is its structure and what is the protocol? What roles and responsibilities are defined? What are the benefits and liabilities?

Give some examples on the type of translation that an adapter can make between the client and the adaptee.

21.6 Further Exercises

Exercise 21.2:

Consider a cooling system for a refrigerating store. The system monitors the temperature using sensors in the storage room and turns cooling on and off. The cooling algorithm reads the temperature via a TemperatureSensor interface that has a single method

```
/** read temperature in Celsius */
public double readTemperature();
```

Now, an existing refrigerating store wants to use our system but their existing temperature sensors have another interface:

```
/** return present temperature
 * @return the temperature in Fahrenheit; the returned reading
 * is the actual measured value times ten. Example: 212.34
 * Fahrenheit is returned as integer value 2123
 */
public int getValue();
```

Construct an ADAPTER that will allow our software to use the existing sensors.

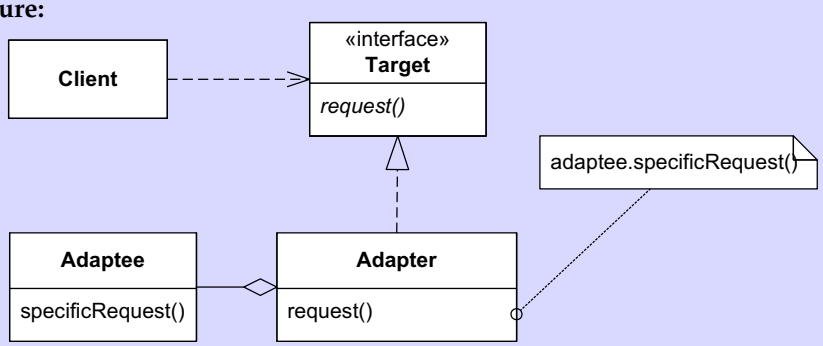
[21.1] Design Pattern: Adapter

Intent Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Problem You have a class with desirable functionality but its interface and/or protocol does not match that of the client needing it.

Solution You put an intermediate object, the adapter, between the client and the class with the desired functionality. The adapter conforms to the interface used by the client and delegate actual computation to the adaptee class, potentially performing parameter, protocol, and return value translations in the process.

Structure:



Roles **Target** encapsulates behavior used by the **Client**. The **Adapter** implements the Target role and delegate actual processing to the **Adaptee** performing parameter and protocol translations in the process.

Cost - Benefit Adapter *lets objects collaborate that otherwise are incompatible*. A single adapter can work with many adaptees—that is, all the adaptee's subclasses.