Clients must be aware of strategies. If the selection of which concrete RateStrategy instance to use is made in the pay station object itself, then this solution is no better than the parametric one: you end up with conditional statements in the pay station production code itself. Thus the pay station object must be told which rate strategy object to use. Thus the client object (typically the one that instantiates the pay station object) has to know about RateStrategy. This can also make it more difficult for a developer to overview the system.

7.7 The Compositional Process

The last proposal, the compositional proposal, is actually an example of using the STRATEGY pattern. Before I describe STRATEGY, let me sum up the line of reasoning that lead to it. The argumentation went along these lines:

- *I identified some behavior that varied.* The rate calculation behavior of the pay station is variable depending on which town the station is located in. Furthermore I can expect this behavior to be variable for new customers that buy the pay station.
- *I stated a responsibility that covered the variable behavior and encapsulated it by expressing it as an interface.* The RateStrategy interface defines the responsibility to *calculate parking time* by defining the method calculateTime.
- *I get the full pay station behavior by delegating the rate calculation responsibility to a delegate.* Now the pay station provides its full behavior as a result of collaboration between itself and an object specializing in rate calculations, either an instance of LinearRateStrategy or ProgressiveRateStrategy.

I will call this three step line of reasoning the ③-①-② process. The numbers refer to three principles for flexible design that were originally stated in the introduction to the first book on design patterns, *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al. 1995). In this book the principles are numbered 1–3 but they are applied in another order: The ③-①-② process first applies the third principle (find variability), then the first (use interface), and finally the second (delegate), hence the reason that I have chosen this odd numbering. I will discuss these principles in great detail in Chapter 16.

This collaboration mind set is a fruitful one in object-oriented design and is actually familiar to everybody working in a company, a public institution, or living in a family. Work is done by coordinating the collaboration of different people with different competences and skills. One person cannot perform all tasks but has to delegate responsibility and work to others. A person having too many concrete tasks to do leads to stress, making errors, and poor performance. *Objects are no different...*

7.8 The Strategy Pattern

The ③-①-② process and the ideas of identifying, encapsulating, and delegating variable behavior has resulted in a design with a number of desirable properties—and

Copyrighted Material. Do Not Distribute.

some liabilities of course. The resulting design is actually an example of the design pattern STRATEGY. STRATEGY is a pattern that addresses the problem of encapsulating a family of algorithms or business rules, and allows implementations of these algorithms to vary independently from the client that uses them. In our case, the business rule is calculation of rates.

The STRATEGY pattern's properties are summarized in the design pattern box 7.1 on page 26. The format used is explained in more detail in Chapter 9. An important aspect of any design pattern is the list of benefits and liabilities of using it to solve your particular design problem. The design pattern box only lists a few keywords but the discussion above is of course the comprehensive version of this.

7.9 Summary of Key Concepts

Often behavior in software systems must come in different variants. The requirements for these variants may stem from customers that have special needs (like the new customer of our pay station system), from us wanting to be able to run (and sell) a system on various operating systems or using various hardware and software configurations, or from the development team itself that needs to execute the system in "testing mode" and/or without actual hardware connected. The points in the production code that must exhibit variable behavior in different variants are called *variability points*. Variants can be handled in different ways but they are all basically variations over four different themes:

- *Source code copy solution.* You copy parts of or the entire software production code and simply replace the code in the variability points.
- *Parametric solution.* You enter conditional statements around the variability points. The conditions branch on a configuration parameter identifying the configuration.
- *Polymorphic solution.* You encapsulate the variability points in instance methods. These can then be overridden in subclasses, one for each required variant.
- *Compositional solution.* You encapsulate the variability points in a well-defined interface and use delegation to compose the overall behavior. Concrete classes, implementing the interface, define each variant's behavior.

Often compositional solutions arise from a line of design thinking that is called the ③-①-② process in this book. It consists of three steps: first you *identify some behavior that needs to vary.* You next abstract the behavior into *a responsibility that covers the variable behavior and express it as an interface.* Finally, you *use delegation to compose the full behavior.*

The STRATEGY pattern is a compositional solution to the problem of supporting variations in algorithms or business rules. The algorithm is encapsulated in an interface and the client delegates to implementations of this interface.

The first comprehensive overview of design patterns was the seminal book by Gamma et al. (1995). The *intent* sections of design pattern boxes in this book are in most cases literate copies from this book and reproduced with permission. The ③-①-② process is inspired by Shalloway and Trott (2004).

Copyrighted Material. Do Not Distribute.

7.10 Selected Solutions

Discussion of Exercise 7.3:

The problem is that there are no elegant solutions! I will discuss a set of potential solutions in detail in Chapter 11 but all of them turn out to be rather clumsy.

Discussion of Exercise 7.4:

As the calculation of rates is delegated to an instance implementing the RateStrategy interface, all you have to do is to change the stored reference to refer to a new instance that implements another rate calculation algorithm.

7.11 Review Questions

Outline the four different proposals to support two products with varying rate structure: What is the idea, what coding is involved? Explain what a variability point is.

What are the benefits and liabilities of the source tree copy proposal? The parametric proposal? The polymorphic proposal? The compositional proposal?

What is *change by addition*? What is *change by modification*?

What are the three steps, ③ ① and ②, involved in the compositional proposal process?

What is the STRATEGY pattern? What problem does it address and what solution does it suggest? What are the objects and interfaces involved? What are the benefits and liabilities?

7.12 Further Exercises

Exercise 7.5:

The reputation of our reliable and flexible pay station has reached Europe and Denmark and a Danish county wants to buy it. However, it should accept Danish coin values and of course use an appropriate rate structure. The requirements are:

- Danish coins have values: 1, 2, 5, 10, and 20 Danish kroner.
- 1 Danish krone should equal 7 minutes parking time.

1. Use the ③ ① ② process to identify and design a compositional solution to the coin type aspect that needs to vary in our pay station product.
2. Another Danish county wants also to buy our system but they want another rate policy: 1 Danish krone should equal 6 minutes parking time, however if the car stays for more than 2 hours then a krone only buys 5 minutes parking time. Analyze if this requirement will affect any code that was introduced by the above requirement. Can the two types of requirements, rate policy and coin validation, be varied independent of each other?

Copyrighted Material. Do Not Distribute.

Exercise 7.6:

A software shop wants to produce a calendar application for the international market. One important requirement is the ability for the calendar to mark public holidays for the given country in the calendar, for instance by giving public holidays a different background color in a graphical week overview.

Sketch how this requirement could be handled by the four different proposals for handling variability. Consider that the calendar must handle at least US, UK, French, and Danish public holidays, and allow easy integration of other nations' public holidays.

Copyrighted Material. Do Not Distribute.

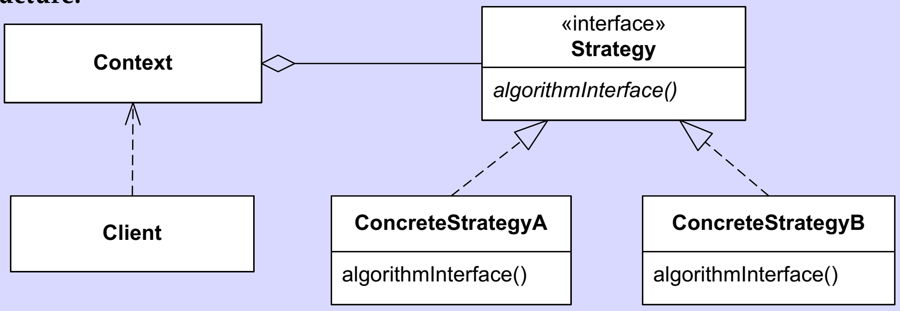
[7.1] Design Pattern: Strategy

Intent Define a family of business rules or algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use it.

Problem Your product must support variable algorithms or business rules and you want a flexible and reliable way of controlling the variability.

Solution Separate the selection of algorithm from its implementation by expressing the algorithm's responsibilities in an interface and let each implementation of the algorithm realize this interface.

Structure:



Roles **Strategy** specifies the responsibility and interface of the algorithm. **ConcreteStrategies** defines concrete behavior fulfilling the responsibility. **Context** performs its work for **Client** by delegating to an instance of type **Strategy**.

Cost - Benefit The benefits are: *Strategies eliminate conditional statements*. It is an *alternative to subclassing*. It facilitates *separate testing* of **Context** and **ConcreteStrategy**. Strategies may be changed at run-time (if they are stateless).
The liabilities are: *Increased number of objects*. *Clients must be aware of strategies*.

Copyrighted Material. Do Not Distribute.