STRATEGY

Object Behavioral

Intent

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

Also Known As

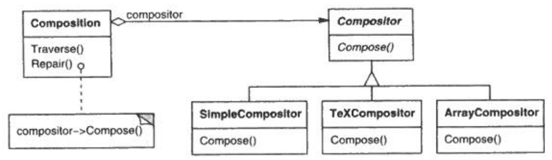
Policy

Motivation

Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:

- Clients that need linebreaking get more complex if they include the linebreaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.
- Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.
- It's difficult to add new algorithms and vary existing ones when linebreaking is an integral part of a client.

We can avoid these problems by defining classes that encapsulate different linebreaking algorithms. An algorithm that's encapsulated in this way is called a **strategy**.



Suppose a Composition class is responsible for maintaining and updating the linebreaks of text displayed in a text viewer. Linebreaking strategies aren't implemented by the class Composition. Instead, they are implemented separately by subclasses of the abstract Compositor class. Compositor subclasses implement different strategies:

- **SimpleCompositor** implements a simple strategy that determines linebreaks one at a time.
- **TeXCompositor** implements the TEX algorithm for finding linebreaks. This strategy tries to optimize linebreaks globally, that is, one paragraph at a time.
- **ArrayCompositor** implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.

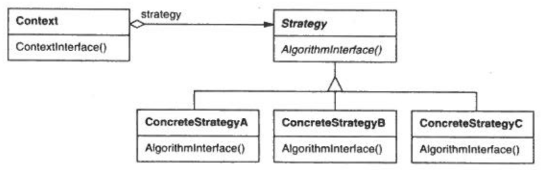
A Composition maintains a reference to a Compositor object. Whenever a Composition reformats its text, it forwards this responsibility to its Compositor object. The client of Composition specifies which Compositor should be used by installing the Compositor it desires into the Composition.

Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms [HO87].
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure



Participants

- **Strategy** (Compositor)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - implements the algorithm using the Strategy interface.
- **Context** (Composition)
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

Collaborations

- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences

The Strategy pattern has the following benefits and drawbacks:

1. *Families of related algorithms.* Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.
2. *An alternative to subclassing.* Inheritance offers another way to support a variety of algorithms or behaviors. You can subclass a Context class directly to give it different behaviors. But this hard-wires the behavior into Context. It mixes the algorithm implementation with Context's, making Context harder to understand, maintain, and extend. And you can't vary the algorithm dynamically. You wind up with many related classes whose only difference is the algorithm or behavior they employ. Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.
3. *Strategies eliminate conditional statements.* The Strategy pattern offers an alternative to conditional statements for selecting desired behavior. When different behaviors are lumped into one class, it's hard to avoid using conditional

statements to select the right behavior. Encapsulating the behavior in separate Strategy classes eliminates these conditional statements.

For example, without strategies, the code for breaking text into lines could look like

```
void Composition::Repair () {
    switch (_breakingStrategy) {
    case SimpleStrategy:
        ComposeWithSimpleCompositor();
        break;
    case TeXStrategy:
        ComposeWithTeXCompositor();
        break;
    // ...
    }
    // merge results with existing composition, if necessary
}
```

The Strategy pattern eliminates this case statement by delegating the line-breaking task to a Strategy object:

```
void Composition::Repair () {
    _compositor->Compose();
    // merge results with existing composition, if necessary
}
```

Code containing many conditional statements often indicates the need to apply the Strategy pattern.

4. *A choice of implementations.* Strategies can provide different implementations of the *same* behavior. The client can choose among strategies with different time and space trade-offs.

5. *Clients must be aware of different Strategies.* The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues. Therefore you should use the Strategy pattern only when the variation in behavior is relevant to clients.

6. *Communication overhead between Strategy and Context.* The Strategy interface is shared by all ConcreteStrategy classes whether the algorithms they implement are trivial or complex. Hence it's likely that some ConcreteStrategies won't use all the information passed to them through this interface; simple ConcreteStrategies may use none of it! That means there will be times when the context creates and initializes parameters that never get used. If this is an issue, then you'll need tighter coupling between Strategy and Context.

7. *Increased number of objects.* Strategies increase the number of objects in an application. Sometimes you can reduce this overhead by implementing strategies as stateless objects that contexts can share. Any residual state is maintained by the context, which passes it in each request to the Strategy

object. Shared strategies should not maintain state across invocations. The Flyweight (195) pattern describes this approach in more detail.

Implementation

Consider the following implementation issues:

1. *Defining the Strategy and Context interfaces.* The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.

   One approach is to have Context pass data in parameters to Strategy operations—in other words, take the data to the strategy. This keeps Strategy and Context decoupled. On the other hand, Context might pass data the Strategy doesn't need.

   Another technique has a context pass *itself* as an argument, and the strategy requests data from the context explicitly. Alternatively, the strategy can store a reference to its context, eliminating the need to pass anything at all. Either way, the strategy can request exactly what it needs. But now Context must define a more elaborate interface to its data, which couples Strategy and Context more closely.

   The needs of the particular algorithm and its data requirements will determine the best technique.

2. *Strategies as template parameters.* In C++ templates can be used to configure a class with a strategy. This technique is only applicable if (1) the Strategy can be selected at compile-time, and (2) it does not have to be changed at run-time. In this case, the class to be configured (e.g., `Context`) is defined as a template class that has a `Strategy` class as a parameter:

   ```
   template <class AStrategy>
   class Context {
       void Operation() { theStrategy.DoAlgorithm(); }
       // ...
   private:
       AStrategy theStrategy;
   };
   ```

   The class is then configured with a `Strategy` class when it's instantiated:

   ```
   class MyStrategy {
   public:
       void DoAlgorithm();
   };

   Context<MyStrategy> aContext;
   ```

   With templates, there's no need to define an abstract class that defines the interface to the `Strategy`. Using `Strategy` as a template parameter also lets you bind a `Strategy` to its `Context` statically, which can increase efficiency.

3. *Making Strategy objects optional.* The Context class may be simplified if it's meaningful *not* to have a Strategy object. Context checks to see if it has a Strategy object before accessing it. If there is one, then Context uses it normally. If there isn't a strategy, then Context carries out default behavior. The benefit of this approach is that clients don't have to deal with Strategy objects at all *unless* they don't like the default behavior.

Sample Code

We'll give the high-level code for the Motivation example, which is based on the implementation of Composition and Compositor classes in InterViews [LCI+92].

The `Composition` class maintains a collection of `Component` instances, which represent text and graphical elements in a document. A composition arranges component objects into lines using an instance of a `Compositor` subclass, which encapsulates a linebreaking strategy. Each component has an associated natural size, stretchability, and shrinkability. The stretchability defines how much the component can grow beyond its natural size; shrinkability is how much it can shrink. The composition passes these values to a compositor, which uses them to determine the best location for linebreaks.

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;    // the list of components
    int _componentCount;       // the number of components
    int _lineWidth;            // the Composition's line width
    int* _lineBreaks;          // the position of linebreaks
                               // in components
    int _lineCount;            // the number of lines
};
```

When a new layout is required, the composition asks its compositor to determine where to place linebreaks. The composition passes the compositor three arrays that define natural sizes, stretchabilities, and shrinkabilities of the components. It also passes the number of components, how wide the line is, and an array that the compositor fills with the position of each linebreak. The compositor returns the number of calculated breaks.

The `Compositor` interface lets the composition pass the compositor all the information it needs. This is an example of "taking the data to the strategy":

```
class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();
};
```

Note that `Compositor` is an abstract class. Concrete subclasses define specific linebreaking strategies.

The composition calls its compositor in its `Repair` operation. `Repair` first initializes arrays with the natural size, stretchability, and shrinkability of each component (the details of which we omit for brevity). Then it calls on the compositor to obtain the linebreaks and finally lays out the components according to the breaks (also omitted):

```
void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;

    // prepare the arrays with the desired component sizes
    // ...

    // determine where the breaks are:
    int breakCount;
    breakCount = _compositor->Compose(
        natural, stretchability, shrinkability,
        componentCount, _lineWidth, breaks
    );

    // lay out components according to breaks
    // ...
}
```

Now let's look at the `Compositor` subclasses. `SimpleCompositor` examines components a line at a time to determine where breaks should go:

```
class SimpleCompositor : public Compositor {
public:
    SimpleCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

`TeXCompositor` uses a more global strategy. It examines a *paragraph* at a time, taking into account the components' size and stretchability. It also tries to give an even "color" to the paragraph by minimizing the whitespace between components.

```
class TeXCompositor : public Compositor {
public:
    TeXCompositor();

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

`ArrayCompositor` breaks the components into lines at regular intervals.

```
class ArrayCompositor : public Compositor {
public:
    ArrayCompositor(int interval);

    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
```

These classes don't use all the information passed in `Compose`. `SimpleCompositor` ignores the stretchability of the components, taking only their natural widths into account. `TeXCompositor` uses all the information passed to it, whereas `ArrayCompositor` ignores everything.

To instantiate `Composition`, you pass it the compositor you want to use:

```
Composition* quick = new Composition(new SimpleCompositor);
Composition* slick = new Composition(new TeXCompositor);
Composition* iconic = new Composition(new ArrayCompositor(100));
```

`Compositor`'s interface is carefully designed to support all layout algorithms that subclasses might implement. You don't want to have to change this interface with every new subclass, because that will require changing existing subclasses. In general, the Strategy and Context interfaces determine how well the pattern achieves its intent.

Known Uses

Both ET++ [WGM88] and InterViews use strategies to encapsulate different line-breaking algorithms as we've described.

In the RTL System for compiler code optimization [JML92], strategies define different register allocation schemes (RegisterAllocator) and instruction set scheduling policies (RISCscheduler, CISCscheduler). This provides flexibility in targeting the optimizer for different machine architectures.

The ET++SwapsManager calculation engine framework computes prices for different financial instruments [EG92]. Its key abstractions are Instrument and YieldCurve. Different instruments are implemented as subclasses of Instrument. YieldCurve calculates discount factors, which determine the present value of future cash flows. Both of these classes delegate some behavior to Strategy objects. The framework provides a family of ConcreteStrategy classes for generating cash flows, valuing swaps, and calculating discount factors. You can create new calculation engines by configuring Instrument and YieldCurve with the different ConcreteStrategy objects. This approach supports mixing and matching existing Strategy implementations as well as defining new ones.

The Booch components [BV90] use strategies as template arguments. The Booch collection classes support three different kinds of memory allocation strategies: managed (allocation out of a pool), controlled (allocations/deallocations are protected by locks), and unmanaged (the normal memory allocator). These strategies are passed as template arguments to a collection class when it's instantiated. For example, an UnboundedCollection that uses the unmanaged strategy is instantiated as `UnboundedCollection<MyItemType*, Unmanaged>`.

RApp is a system for integrated circuit layout [GA89, AG90]. RApp must lay out and route wires that connect subsystems on the circuit. Routing algorithms in RApp are defined as subclasses of an abstract Router class. Router is a Strategy class.

Borland's ObjectWindows [Bor94] uses strategies in dialogs boxes to ensure that the user enters valid data. For example, numbers might have to be in a certain range, and a numeric entry field should accept only digits. Validating that a string is correct can require a table look-up.

ObjectWindows uses Validator objects to encapsulate validation strategies. Validators are examples of Strategy objects. Data entry fields delegate the validation strategy to an optional Validator object. The client attaches a validator to a field if validation is required (an example of an optional strategy). When the dialog is closed, the entry fields ask their validators to validate the data. The class library provides validators for common cases, such as a RangeValidator for numbers. New client-specific validation strategies can be defined easily by subclassing the Validator class.

Related Patterns

Flyweight (195): Strategy objects often make good flyweights.