Strategy

Intent

Define a family of algorithms, encapsulate each one, and make theminterchangeable. Strategy lets the algorithm vary independently fromclients that use it.

Also Known As

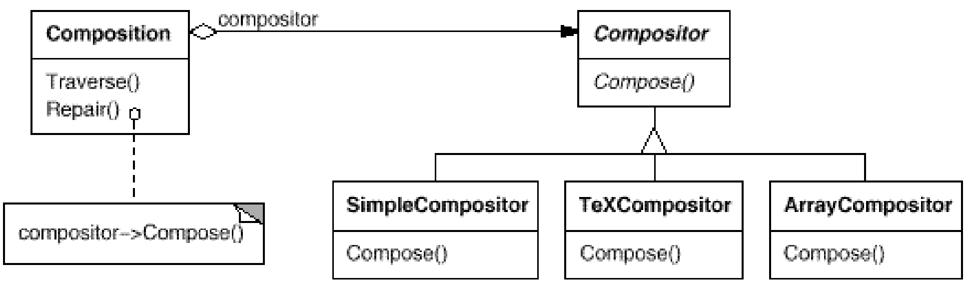
Policy

Motivation

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

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

We can avoid these problems by defining classes that encapsulatedifferent linebreaking algorithms. An algorithm that's encapsulated inthis way is called a **strategy**.



Suppose a Composition class is responsible for maintaining andupdating 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 abstractCompositor class. Compositor subclasses implement different strategies:

- **SimpleCompositor**implements a simple strategy that determines linebreaks one at atime.
- **TeXCompositor**implements the TeX algorithm for finding linebreaks. This strategytries to optimize linebreaks globally, that is, one paragraph at atime.
- **ArrayCompositor**implements a strategy that selects breaks so that each row has a fixednumber of items. It's useful for breaking a collection of icons intorows, for example.

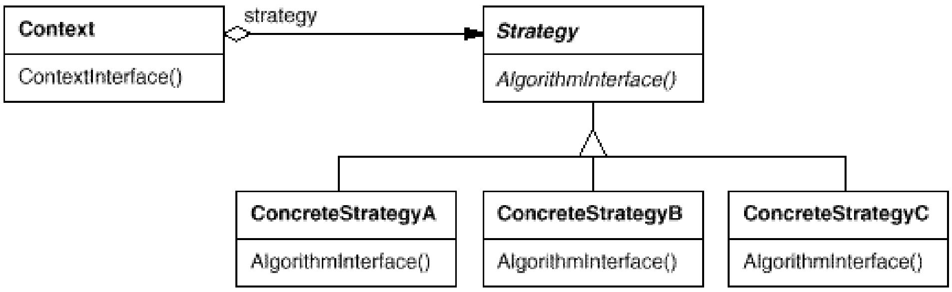
A Composition maintains a reference to a Compositor object. Whenever aComposition reformats its text, it forwards this responsibility to itsCompositor object. The client of Composition specifies whichCompositor should be used by installing the Compositor it desires intothe Composition.

Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategiesprovide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might definealgorithms reflecting different space/time trade-offs.Strategies can be used when these variants are implemented as a classhierarchy of algorithms [HO87].
- an algorithm uses data that clients shouldn't know about. Use theStrategy pattern to avoid exposing complex, algorithm-specific datastructures.
- a class defines many behaviors, and these appear as multipleconditional statements in its operations. Instead of manyconditionals, move related conditional branches into their ownStrategy 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. Acontext may pass all data required by the algorithm to the strategywhen the algorithm is called. Alternatively, the context can passitself as an argument to Strategy operations. That lets the strategycall back on the context as required.
- A context forwards requests from its clients to its strategy. Clientsusually create and pass a ConcreteStrategy object to the context;thereafter, clients interact with the context exclusively. There isoften a family of ConcreteStrategy classes for a client to choosefrom.

Consequences

The Strategy pattern has the following benefits and drawbacks:

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

   For example, without strategies, the code for breakingtext 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 thelinebreaking task to a Strategy object:

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

Code containing many conditional statements often indicatesthe 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 differenttime and space trade-offs.
5. *Clients must be aware of different Strategies.*The pattern has a potential drawback in that a client must understandhow Strategies differ before it can select the appropriate one.Clients might be exposed to implementation issues. Therefore youshould use the Strategy pattern only when the variation in behavior isrelevant to clients.
6. *Communication overhead between Strategy and Context.*The Strategy interface is shared by all ConcreteStrategy classeswhether the algorithms they implement are trivial or complex. Henceit's likely that some ConcreteStrategies won't use all the informationpassed to them through this interface; simple ConcreteStrategies mayuse none of it! That means there will be times when the contextcreates and initializes parameters that never get used. If this is anissue, then you'll need tighter coupling between Strategy and Context.
7. *Increased number of objects.*Strategies increase the number of objects in an application. Sometimesyou can reduce this overhead by implementing strategies as statelessobjects that contexts can share. Any residual state is maintained by thecontext, which passes it in each request to the Strategy object. Sharedstrategies should not maintain state across invocations. The Flyweight (218) pattern describes this approach in moredetail.

Implementation

Consider the following implementation issues:

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

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

   Another technique has a context pass *itself* as an argument, andthe 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, thestrategy can request exactly what it needs. But now

Context mustdefine a more elaborate interface to its data, which couples Strategyand Context more closely.

The needs of the particular algorithm and its data requirements willdetermine 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 selectedat 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) isdefined as a template class that has a Strategy class as aparameter:

```
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 atemplate parameter also lets you bind a Strategy to itsContext statically, which can increase efficiency.

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

Sample Code

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

The Composition class maintains a collection ofComponent instances, which represent text and graphicalelements in a document. A composition arranges component objects intolines using an instance of a Compositor subclass, whichencapsulates a linebreaking strategy. Each component has anassociated natural size, stretchability, and shrinkability. Thestretchability defines how much the component can grow beyond itsnatural size; shrinkability is how much it can shrink. Thecomposition passes these values to a compositor, which uses them todetermine 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 todetermine where to place linebreaks. The composition passes thecompositor three arrays that define natural sizes, stretchabilities,and shrinkabilities of the components. It also passes the number ofcomponents, how wide the line is, and an array that the compositorfills with the position of each linebreak. The compositor returns thenumber of calculated breaks.

The Compositor interface lets the composition pass thecompositor 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. Concretesubclasses define specific linebreaking strategies.

The composition calls its compositor in its Repairoperation. Repair first initializes arrays with the naturalsize, stretchability, and shrinkability of each component (the detailsof which we omit for brevity). Then it calls on the compositor toobtain the linebreaks and finally lays out the components according tothe 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 todetermine 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' sizeand stretchability. It also tries to give an even "color" to theparagraph 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 regularintervals.

```
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 inCompose. SimpleCompositor ignores the stretchabilityof the components, taking only their natural widths into account.TeXCompositor uses all the information passed to it, whereasArrayCompositor ignores everything.

To instantiate Composition, you pass it the compositoryou 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 alllayout algorithms that subclasses might implement. You don't want tohave to change this interface with every new subclass, because that willrequire changing existing subclasses. In general, the Strategy andContext interfaces determine how well the pattern achieves its intent.

Known Uses

Both ET++ [WGM88] and InterViews use strategies to encapsulatedifferent linebreaking 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 theoptimizer for different machine architectures.

The ET++SwapsManager calculation engine framework computes prices fordifferent financial instruments [EG92]. Its keyabstractions are Instrument and YieldCurve. Different instruments areimplemented as subclasses of Instrument. YieldCurve calculatesdiscount factors, which determine the present value of future cashflows. Both of these classes delegate some behavior to Strategyobjects. The framework provides a family of ConcreteStrategy classesfor generating cash flows, valuing swaps, and calculating discountfactors. You can create new calculation engines by configuringInstrument and YieldCurve with the different ConcreteStrategy objects.This approach supports mixing and matching existing Strategyimplementations as well as defining new ones.

The Booch components [BV90] use strategies as templatearguments. The Booch collection classes support three different kinds ofmemory allocation strategies: managed (allocation out of a pool),controlled (allocations/deallocations are protected by locks), andunmanaged (the normal memory allocator). These strategies are passed astemplate arguments to a collection class when it's instantiated. Forexample, an UnboundedCollection that uses the unmanaged strategy isinstantiated as UnboundedCollection.

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

Borland's ObjectWindows [Bor94] uses strategies in dialogsboxes to ensure that the user enters valid data. For example, numbers mighthave to be in a certain range, and a numeric entry field should acceptonly digits. Validating that a string is correct can require atable look-up.

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

Related Patterns

Flyweight (218): Strategy objects often make good flyweights.