Strategy

Stéphane Ducasse
stephane.ducasse@inria.fr
http://stephane.ducasse.free.fr/

Define a family of algorithms, encapsulate each in a separate class and define each class with the same interface so that they can be interchangeable.

Also Know as Policy

Motivation

Many algorithms exist for breaking a stream into lines. Hardwiring them into the classes that requires them has the following problems:

Clients get more complex
Different algorithms can be used at different times
Difficult to add new algorithms at run-time
**Code Smells**

Composition>>repair
formatting == #Simple
  ifTrue: [ self formatWithSimpleAlgo]
  ifFalse: [ formatting == #Tex
            ifTrue: [self formatWithTex]
            ....]

**Alternative**

Composition>>repair
| selector |
selector := ('formatWith, formatting) asSymbol.
self perform: selector

Still your class gets complex...

**Inheritance?**

May not be the solution since:
- you have to create objects of the right class
- it is difficult to change the policy at run-time
- you can get an explosion of classes bloated with the use of a functionality and the functionalities.
- no clear identification of responsibility

**Strategy Solution**
**When**

Many related classes differ only in their behavior. You have variants of an algorithm (space/time). An algorithm uses data that the clients does not have to know.

**Structure**

```
Composition>>repair
formatter format: self
```

**Participants**

- **Strategy (Compositor)**
  declares an interface common to all concrete strategies

- **Concrete Strategies**
  implement algorithm

- **Context**
  configure with concrete strategy
  maintains a reference to the concrete strategy
  may define an interface to let the strategy access data

**Collaborations (i)**

- **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.

  ```
  GraphVisualizer>>graphIt
  ....
  grapher plot: data using: graphPane pen
  ```
Context passes itself as argument

Also know as self-delegation...

GraphVisualizer>>graphIt
grapher plotFor: self

BartChartGrapher>>plotFor: aGraphVisualizer
|data|
data := aGraphVisualizer data

BackPointer

Graperh class>>for: aGraphVisualizer
^ self new graphVisualizer: aGraphVisualizer

BartChartGrapher>>plot
...
graphVisualizer data..
graphVisualizer pen

Collaboration (ii)

“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." GOF

Not sure that the client has to choose...

Consequences

Define a family of pluggable algorithms
Eliminates conditional statements
Clients can choose between several implementations
Clients must be aware of the different strategies
Increase the number of objects
Communication overhead between client and strategies
Weaken encapsulation of the client
Domain-Specific Objects as Strategies

Strategies do not have to be limited to one single algorithm.
They may represent domain specific knowledge.

Mortgage
  FixedRateMortgage
  OneYear...

Known Uses

ImageRenderer in VW: “a technique to render an image using a limited palette”
ImageRenderer
  NearestPaint
  OrderedDither
  ErrorDiffusion

View-Controller
  A view instance uses a controller object to handle and respond to user input via mouse or keyboard.