Strategy Pattern – Behavioural

• Intent

  » Define a family of algorithms, encapsulate each one, and make them interchangeable

  » Let the algorithm vary independently from the clients that use it

• Also known as

  » Policy
Motivation

• Many algorithms exist for breaking a stream of text into lines. Hardwiring all such algorithms is not good

  » Clients that need line breaking get more complex if they include the line breaking algorithms
    > Clients become more difficult to maintain

  » Different algorithms will be appropriate at different times
    > Do not want to support multiple algorithms if all are not used

  » Difficult to add new algorithms and vary existing ones when line breaking is an integral part of a client
Abstract Architecture

```
CONTEXT
context_interface

strategy

STRATEGY
algorithm_interface

CONCRETE_STRATEGY_1
algorithm_interface

CONCRETE_STRATEGY_2
algorithm_interface

CONCRETE_STRATEGY_3
algorithm_interface
```

Strategy-4
Participants

- **Strategy**
  - *Defines an interface common to all supported algorithms*

- **ConcreteStrategy**
  - *Implements the algorithm using the Strategy interface*

- **Context**
  - *Is configured with a ConcreteStrategy object*
  - *Maintains a reference to a Strategy object*
  - *May define an interface that lets Strategy access its data*
Strategy-6

Applicability

» Many related classes differ only in their behaviour
  > Strategies provide a way to configure a class with one of many behaviours

» Need different variants of an algorithm
  > For example – Reflecting different time-space tradeoffs

» An algorithm uses data that clients should not know about

» A class defines many behaviours and these appear as multiple conditional statements in its operation
  > In place of conditionals, move related branches into their own strategy class
Consequences

• Families of related algorithms
  » Hierarchies of Strategy classes define a family of algorithms or behaviours for contexts to reuse

• Alternative to subclassing
  » Using inheritance within Context hardwires behaviour
    > More difficult to design, understand, maintain and extend
Consequences – 2

• Eliminate conditional statements
  » Put conditional clauses into Strategy subclasses
  » Program text using many conditionals may benefit from using Strategy

• Choice of implementation
  » Provide different implementation of the same behaviour
    > Consider time-space tradeoffs
Consequences – 3

• Clients must be aware of different Strategies
  » Clients need to understand different Strategies to be able to pick the appropriate one

• Communication overhead between Strategy and Context
  » All algorithm variations share the same interface
    > With a large variance in size of interfaces needed for variants, can have excessive communication overhead
Consequences – 4

• Increased number of objects
  » Strategy increases the number of objects
  » Can reduce space overhead by using stateless objects that share context

• Do not maintain state across invocations
  » Use Flyweight pattern instead
Related Patterns

• Strategy objects often make good flyweights