Object-Oriented Programming
Strategy Pattern

CSIE Department, NTUT
Woei-Kae Chen
Strategy pattern: Define a family of algorithms, encapsulate each one, and make them interchangeable.
Design patterns

- simple and elegant solutions to specific problems in object-oriented software design
- capture solutions that have been developed and evolved over time
- make your own designs more
  - flexible,
  - modular,
  - reusable, and
  - understandable
Strategy pattern: Intent

- Define a family of algorithms.
  - Encapsulate each one (as an object).
  - Make them interchangeable (through polymorphism).
  - Vary independently from clients that use it

- Also known as: Policy

- Note: OMT ➔ page 365
Strategy pattern: Motivation

Composition

SimpleComposition()
TexComposition()
ArrayComposition()
Strategy pattern: Motivation

- **Hard-wiring** line breaking algorithms
  - Clients get bigger and harder to maintain.
  - Different algorithms will be appropriate at different times
    - All algorithms exist simultaneously
  - Difficult to add new algorithms and vary existing ones.
- **Strategy** avoids these problems
Strategy pattern: Example

Composition
- Traverse ()
- Repair ()

Compositor -> Compose ()

Compositor
- Compose ()

SimpleCompositor
- Compose ()

TexCompositor
- Compose ()

ArrayCompositor
- Compose ()

compositor

line breaking
Strategy pattern: Applicability

- Many related classes differ only in their behavior.
- You need different variants of an algorithm.
- An algorithm uses data that clients shouldn't know about. Avoid exposing complex, algorithm-specific data structure to clients.
- A class defines many behaviors (use of multiple conditional statements).
Strategy pattern: Structure

**Context**
- ContextInterface()

**Strategy**
- AlgorithmInterface()

**Concrete Strategies**
- **ConcreteStrategyA**
  - AlgorithmInterface()
- **ConcreteStrategyB**
  - AlgorithmInterface()
- **ConcreteStrategyC**
  - AlgorithmInterface()

**Client**
Strategy pattern: Participants

- **Strategy (Compositor)**
  - Declares an `interface` common to all supported algorithms.

- **ConcreteStrategy (SimpleCompositor ...)**
  - 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.
Strategy pattern: Collaborations

• Passing data
  – A context may pass all data required by the algorithm.
  – Alternatively, the context can pass itself as an argument to Strategy operations.

• A context forwards requests from its clients to its strategy.
Strategy pattern: Consequences

• Benefits:
  – *Families of related algorithms*
  – *An alternative to subclassing*
    • subclassing context
  – *Strategies eliminate conditional statements.*
    • no switch statements
  – *A choice of implementations*
    • client can choose different strategies with different time and space trade-offs
Strategy pattern: Consequences

- **Drawbacks**
  - *Clients must be aware of different Strategies.*
  - to select the appropriate one
  - *Communication overhead between Strategy and Context.*
  - *Increased number of objects.*
Strategy pattern: Implementation

Defining the Strategy and Context interfaces.

– Context pass data to Strategy operations.
  • Decoupled.
  • Might pass unneeded data.
– Context passed itself as an argument.
  • Strategy requests exactly what it needs.
  • Must define a more elaborate interface.
  • Tightly coupled.
Strategies as template parameters.
- The strategy can be selected at compile-time.
- It does not have to be changed at run-time.
- No abstract Strategy class.

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

Making Strategy objects optional
- context implements a default Strategy
- client select Strategy only when default is no good

```
class MyStrategy {
public:
  void DoAlgorithm();
};
Context<MyStrategy> myStrategyContext;
```
Strategy pattern: Related patterns

- **Flyweight**
  - Strategy objects often make good flyweights.