南京大学
NANJING UNIVERSITY

The Strategy Pattern

Linzhang Wang
Dept. of Computer Sci&Tech,
Nanjing University

SEG
Nanjing University

Intent

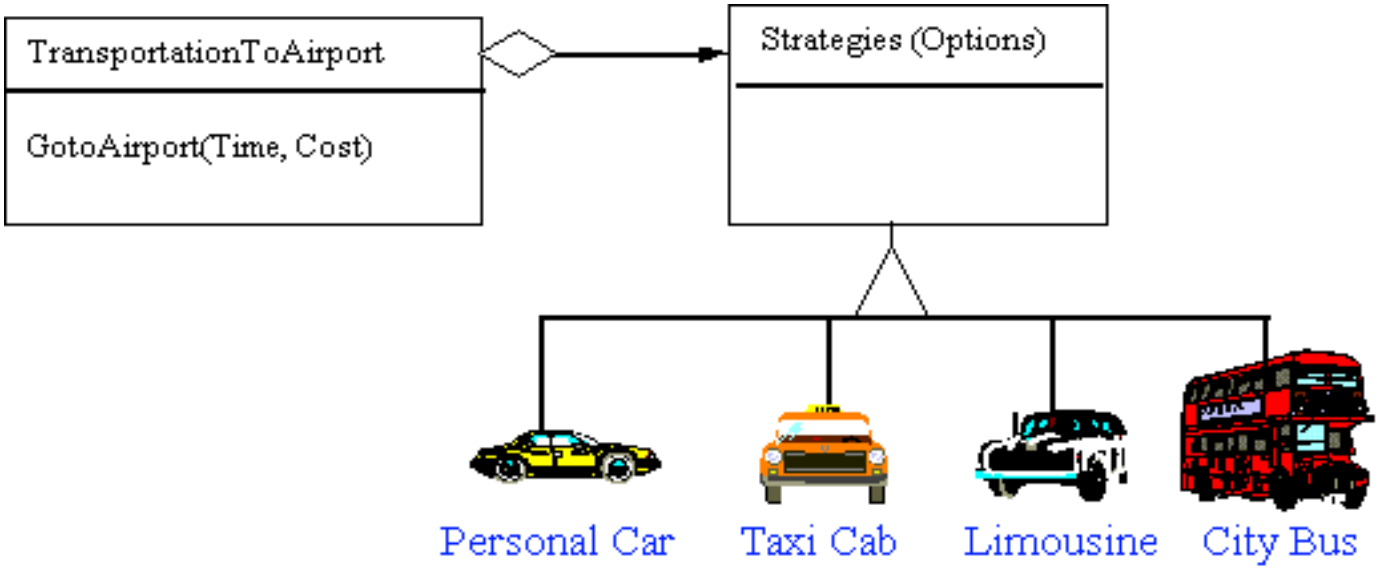
- Define a family of algorithm, encapsulate each one, and make them interchageable.
- Strategy lets the algorithm vary independently from clients that use it.
- Also known as Policy pattern

Motivation

- Hard-wiring all algorithms into a class makes the client more complex, bigger and harder to maintain.
- We do not want to use all the algorithms.
- It's difficult to add and vary algorithms when the algorithm code is an integral part of a client.

Example

Modes of transportation to an airport is an example of a Strategy. Several options exist, such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.

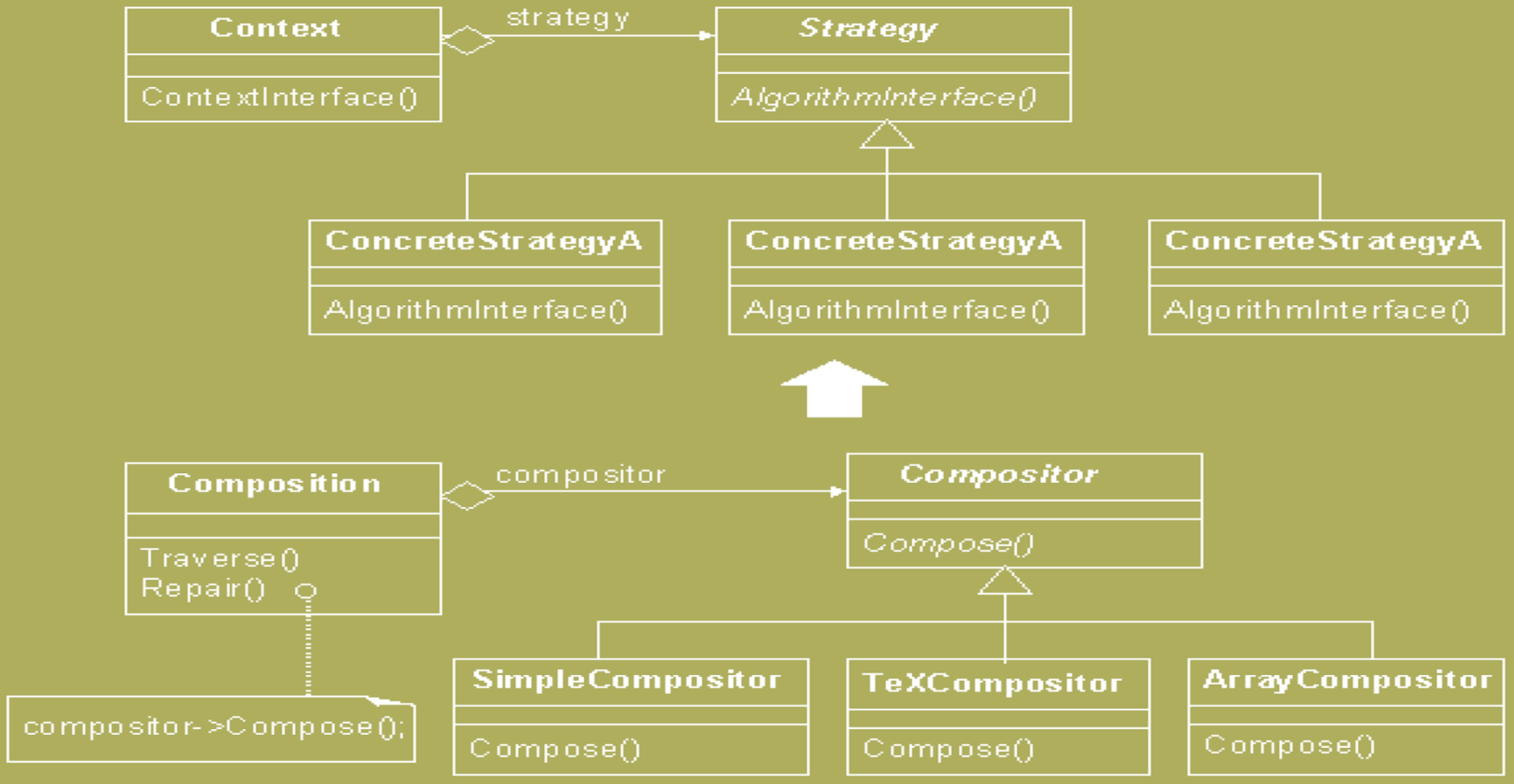


Example

Many algorithms exists for breaking a string into lines.

- Simple Compositor is a simple linebreaking method.
- TeX Compositor uses the TeX linebreaking strategy that tries to optimize linebreaking by breaking one paragraph at a time.
- Array Compositor breaks a fixed number of items into each row.

Structure


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
Composition
Traverse()
Repair()
compositor
Compositor
Compose()
compositor->Compose();
SimpleCompositor
Compose()
TeXCompositor
Compose()
ArrayCompositor
Compose()

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.
  - Algorithms reflecting different space/time trade-offs.
  - Strategies can be used when these variants are implemented as a class hierarchy of algorithms.

Applicability(2)

- An algorithm uses data-structure 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.

Participants

- Strategy(Compositor)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy
  - implements the algorithm using the Strategy interface.

Paritcipants(2)

- 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.

Collaborations(2)

- 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

- 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.

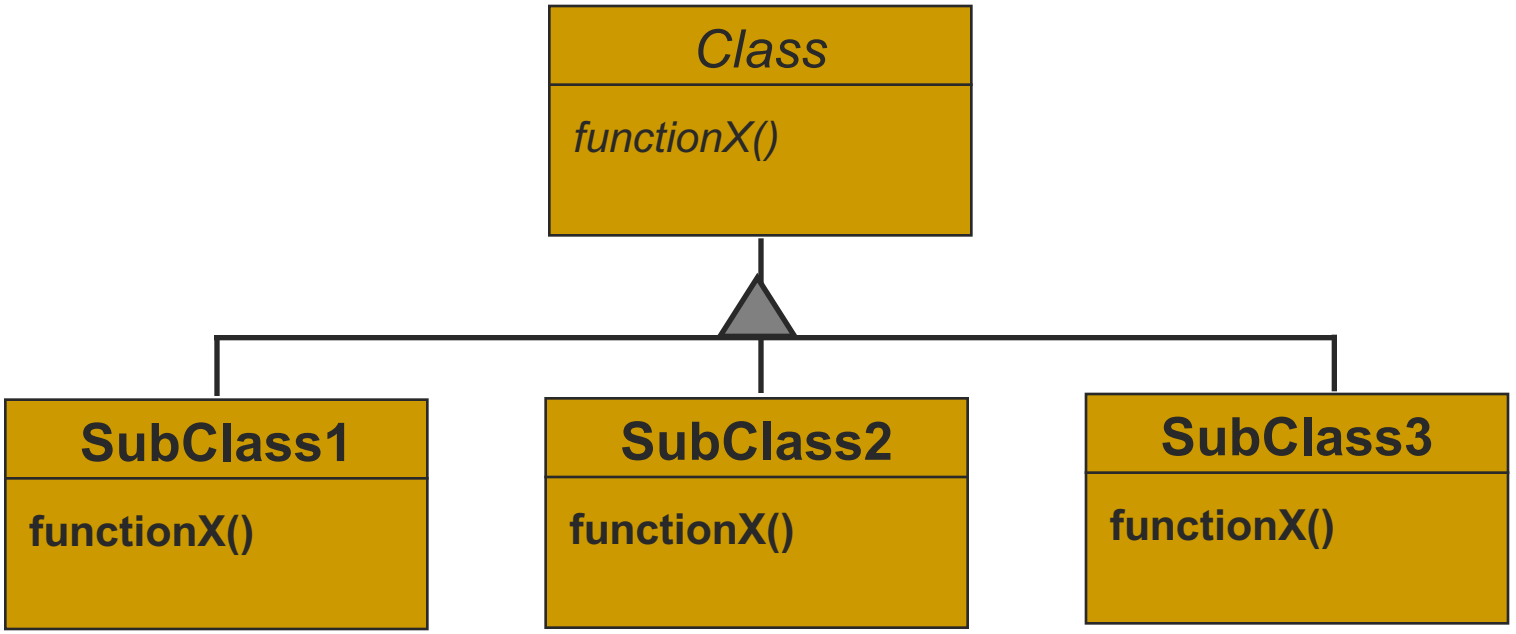
Consequences(2)

- An alternative to subclassing.
  - another way to support a variety of algorithms or behaviors.
  - Subclassing:mixes the algorithm implementation with context's, making Context harder to understand, maintain, and extend. Can not vary the algorithm dynamically.
  - Strategy: vary the algorithm independently of its context, making it easier to switch, understand, and extend.

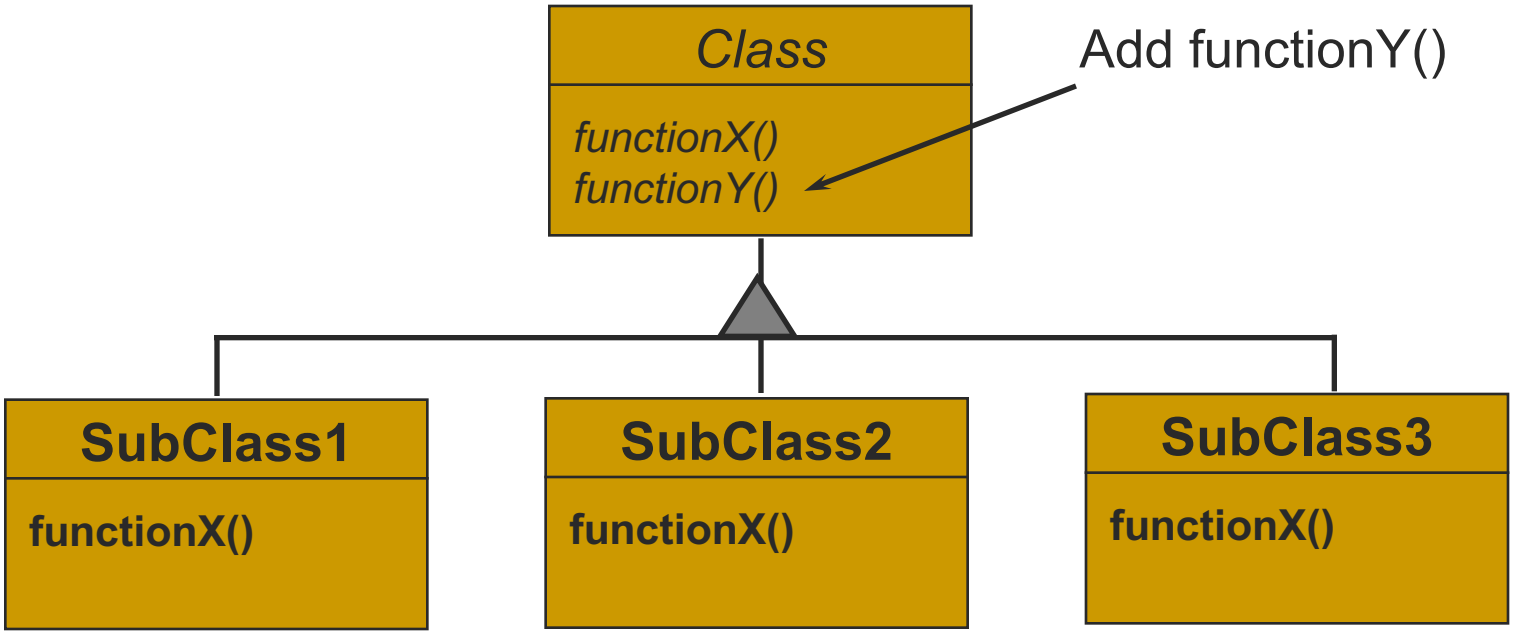
Strategy vs. Subclassing

- Strategy can be used in place of subclassing
- Strategy is more dynamic
- Multiple strategies can be mixed in any combination where subclassing would be difficult

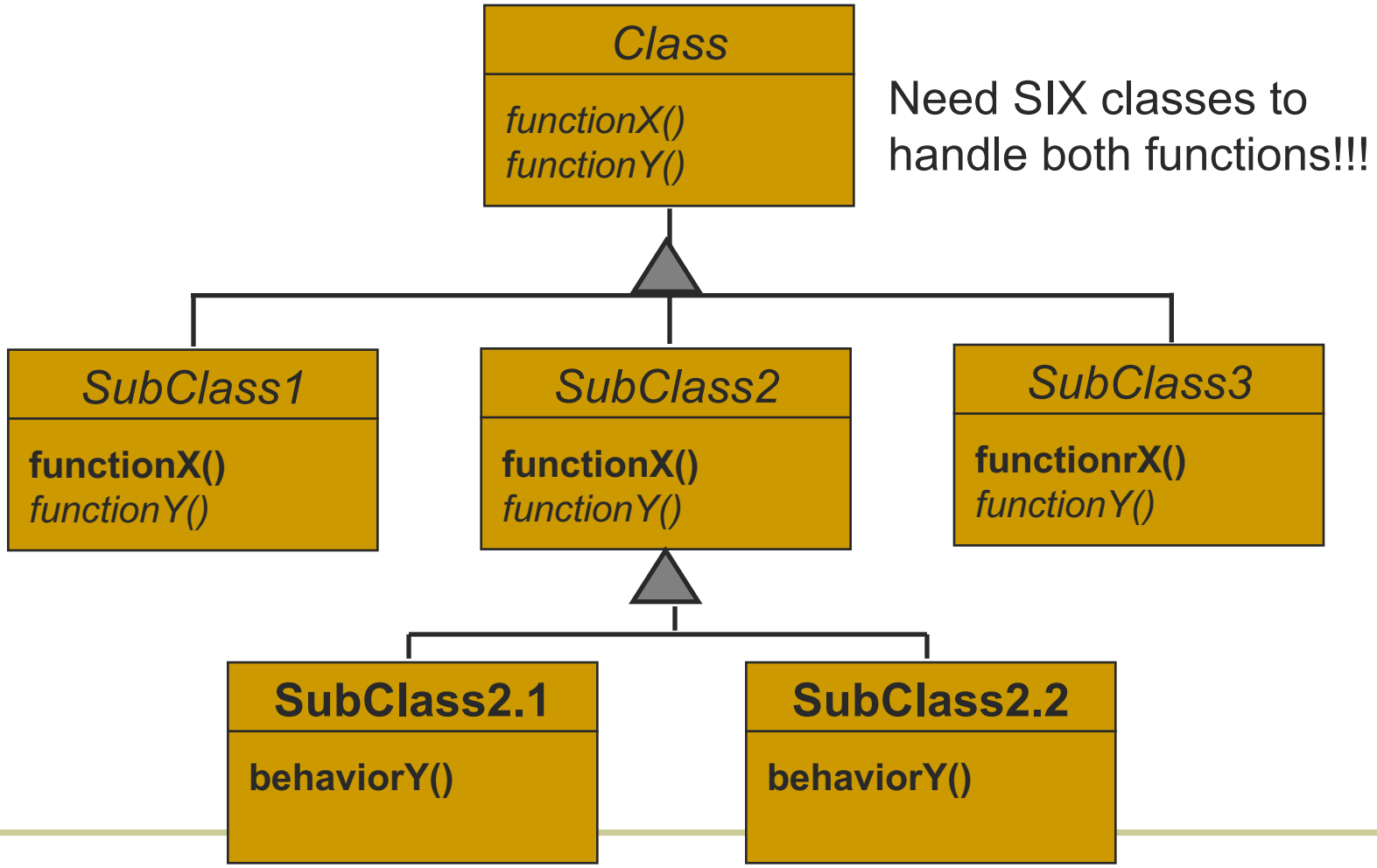
Subclassing



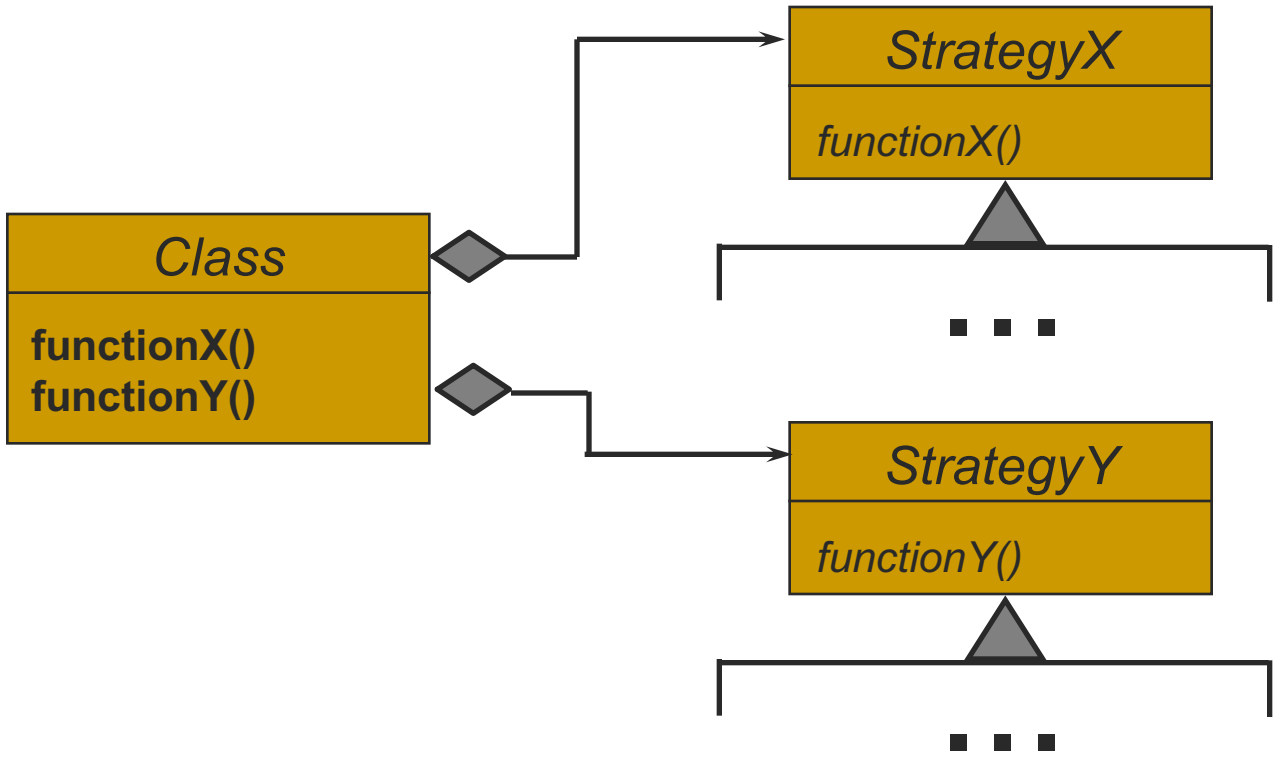
Add a function



What happens?



Strategy makes this easy!



Consequence(3)

- Strategies eliminate conditional statements.
  - When different behaviors are lumped into one class, it's hard to avoid using conditional statements to select the right one.
  - Encapsulating the behavior in separate Strategy classes eliminates these conditional statements.
  - Code containing many conditional statements often indicates the need to apply the Strategy pattern.

Consequence(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.

Consequence(5)

- Client must be aware of different Strategies.
  - A potential drawback: a client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues.
  - You should use Strategy pattern only when the variation in behavior is relevant to Clients.

Consequence(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.
  - Some extra information may be passed to algorithm but not used.
  - If this is an issue, then you'll need tighter coupling between Strategy and Context.

Consequence(7)

- Increased number of objects
  - Strategies increase the number of objects in an application.
  - Sometimes you reduce this overhead by implementing strategies as stateless objects that contexts can share.

Implementation

- Consider the following implementation issues:
  - Defining the Strategy and Context interfaces.
  - Strategies as template parameters.
  - Making Strategy objects optional.

Defining the Strategy and Context interfaces

- The interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.
  - Context pass data in parameters to Strategy operations.
  - 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.) But this couples Strategy and Context closely.

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, Context uses it normally.
  - If there is no one, Context carries out default behavior.