Object-Oriented Programming

Design Pattern Strategy

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Strategy

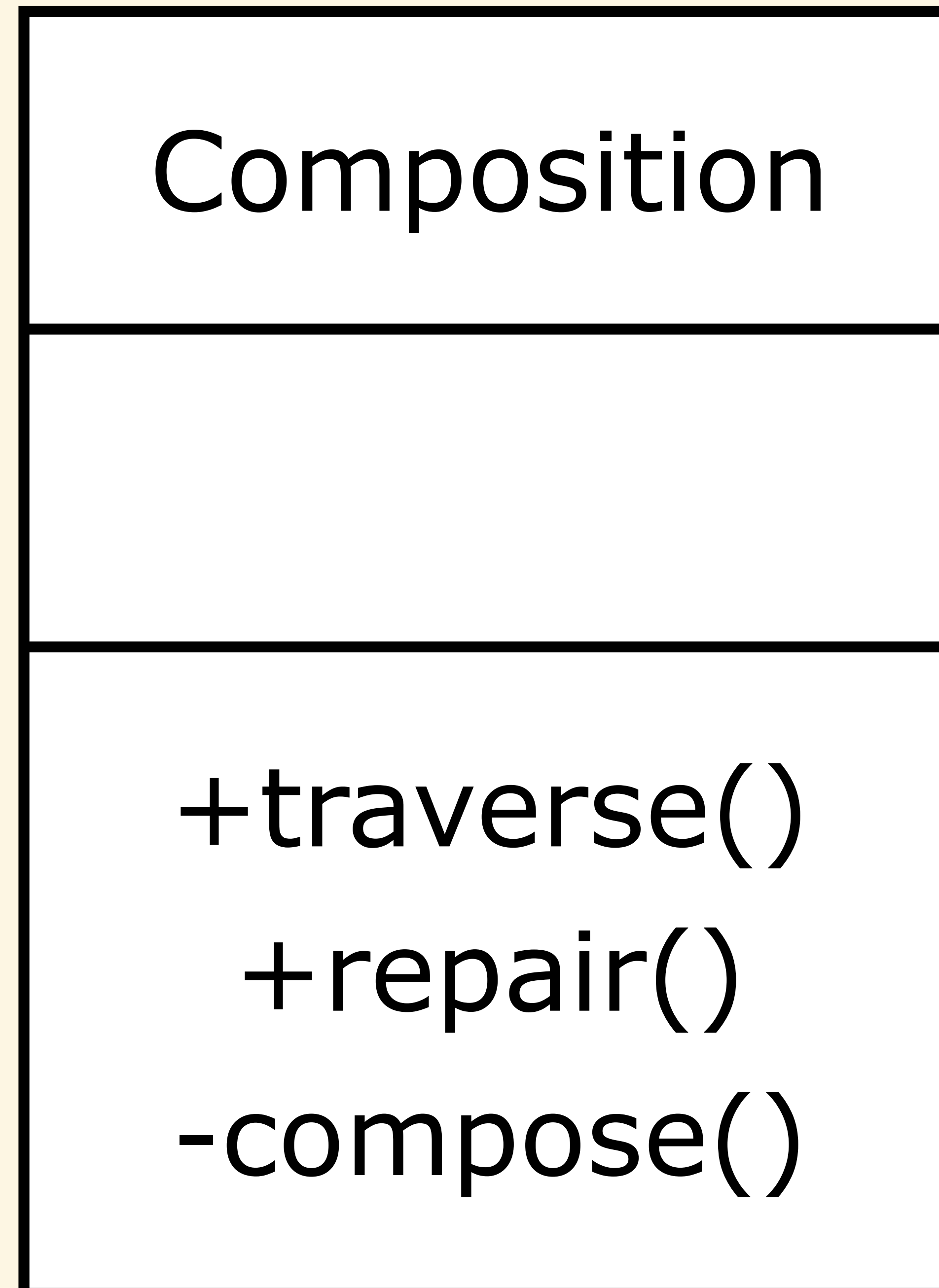
Define a family of algorithms, encapsulate each one, and make them interchangeable

Strategy lets the algorithm vary independently from clients that use it

- Behavioral Pattern

- AKA: Policy

Strategy: Motivation



- Usage: Compose Text
- Needs to be broken into lines
- Hyphenation to create more uniform lines
- Text compositions are quite complex
- Current design only allows a single composition algorithm

Strategy: Motivation Problems

Composition

+traverse()
+repair()
-composeWithSimpleCompositor()
-composeWithTeXCompositor()
--composeWithArrayCompositor()

- Multiple ways to break text into lines
- Hardcoding the algorithms into the Compositor class is difficult
- If you add the line-breaking code into Compositor, it becomes more complex, especially with multiple algorithms
- Different algorithms are appropriate at different times, and we don't want to support them all
- When the Compositor directly contains the line-breaking code, it is difficult to add new algorithms and change existing

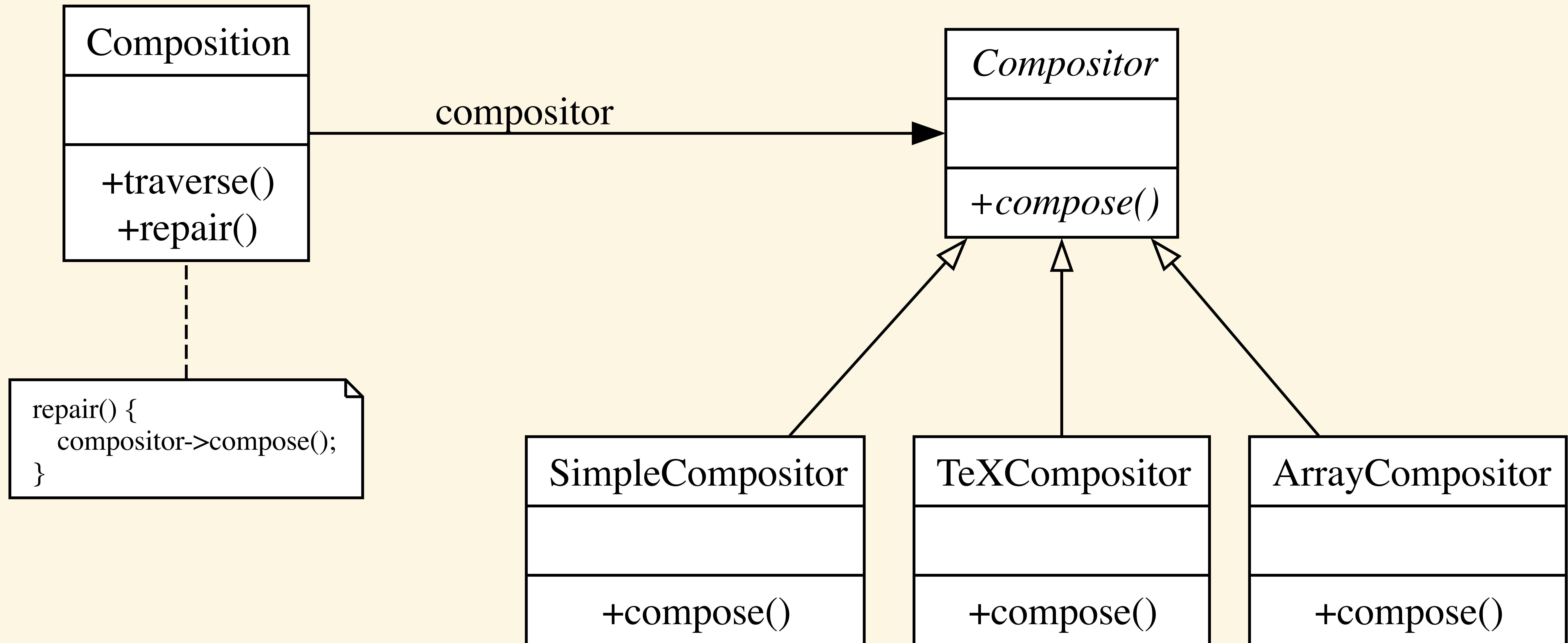
Strategy: Motivation Problems

Composition
+traverse() +repair() -composeWithSimpleCompositor() -composeWithTeXCompositor() --composeWithArrayCompositor()

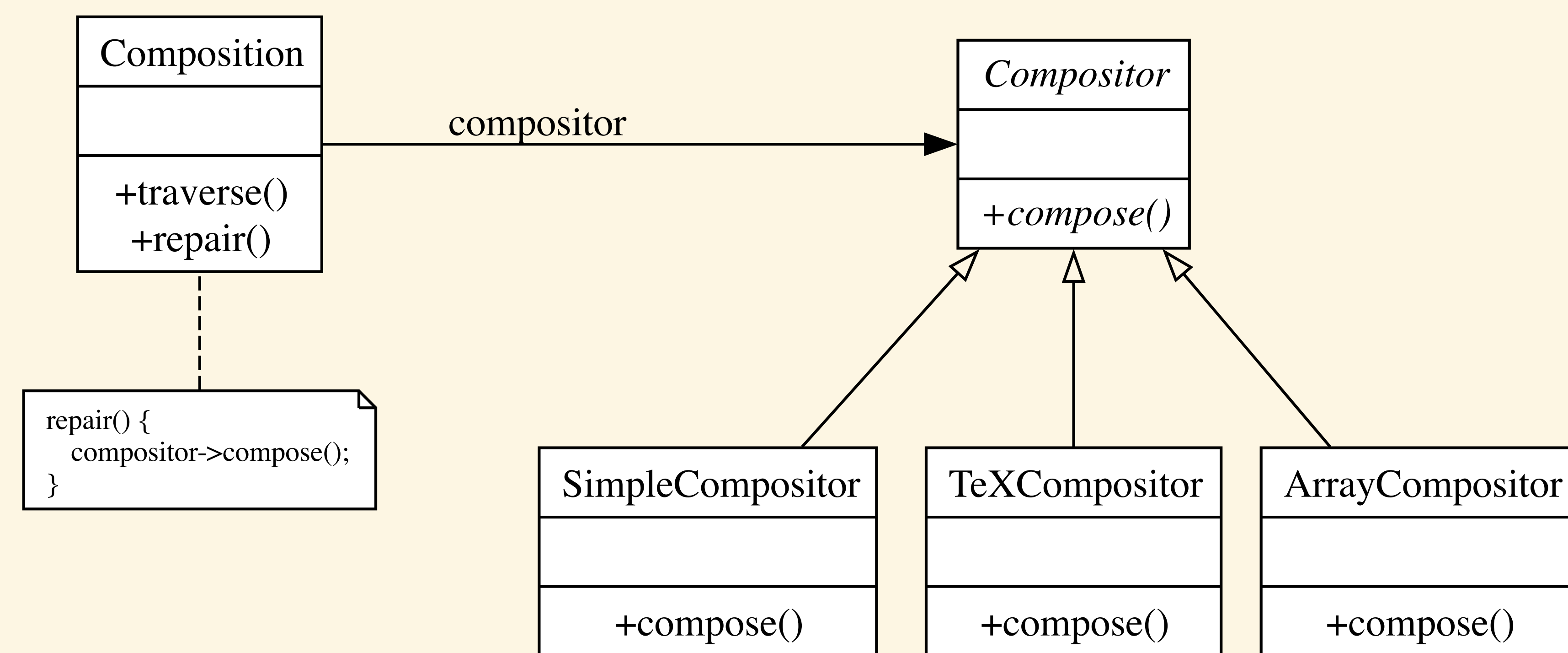
```
void Composition::repair() {
    // ...

    switch (breakingStrategy) {
    case SimpleStrategy:
        composeWithSimpleCompositor();
        break;
    case TeXStrategy:
        composeWithTeXCompositor();
        break;
    case ArrayStrategy:
        composeWithArrayCompositor();
    };
    // ...
}
```

Strategy: Motivation

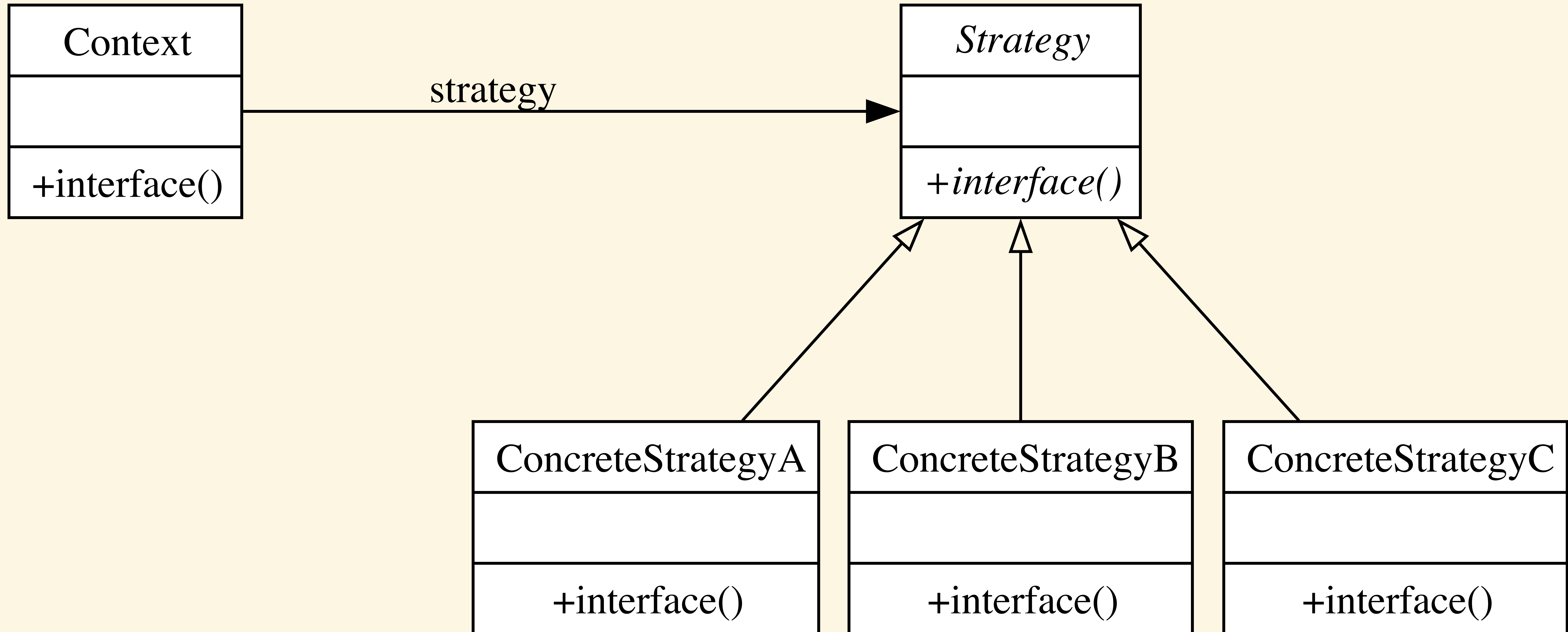


Strategy: Motivation Roles

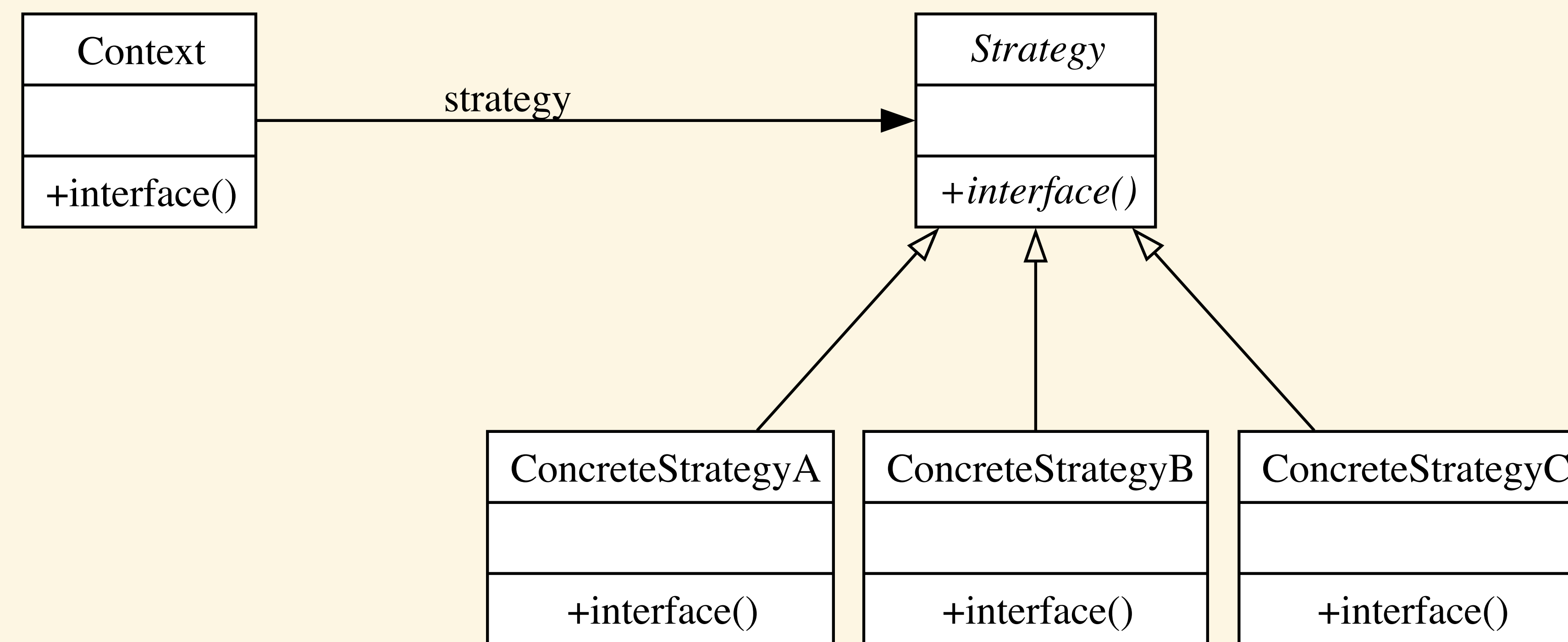


- Composition class - maintains and updates line breaks of text
- Line-breaking strategies are implemented by subclasses of the (abstract) Compositor class
- SimpleCompositor - a simple strategy for determining line breaks
- TeXCompositor - implements the TeX algorithm that optimizes line breaks an entire paragraph at a time
- ArrayCompositor - implements a strategy so that each row has a fixed number of items

Strategy: Structure

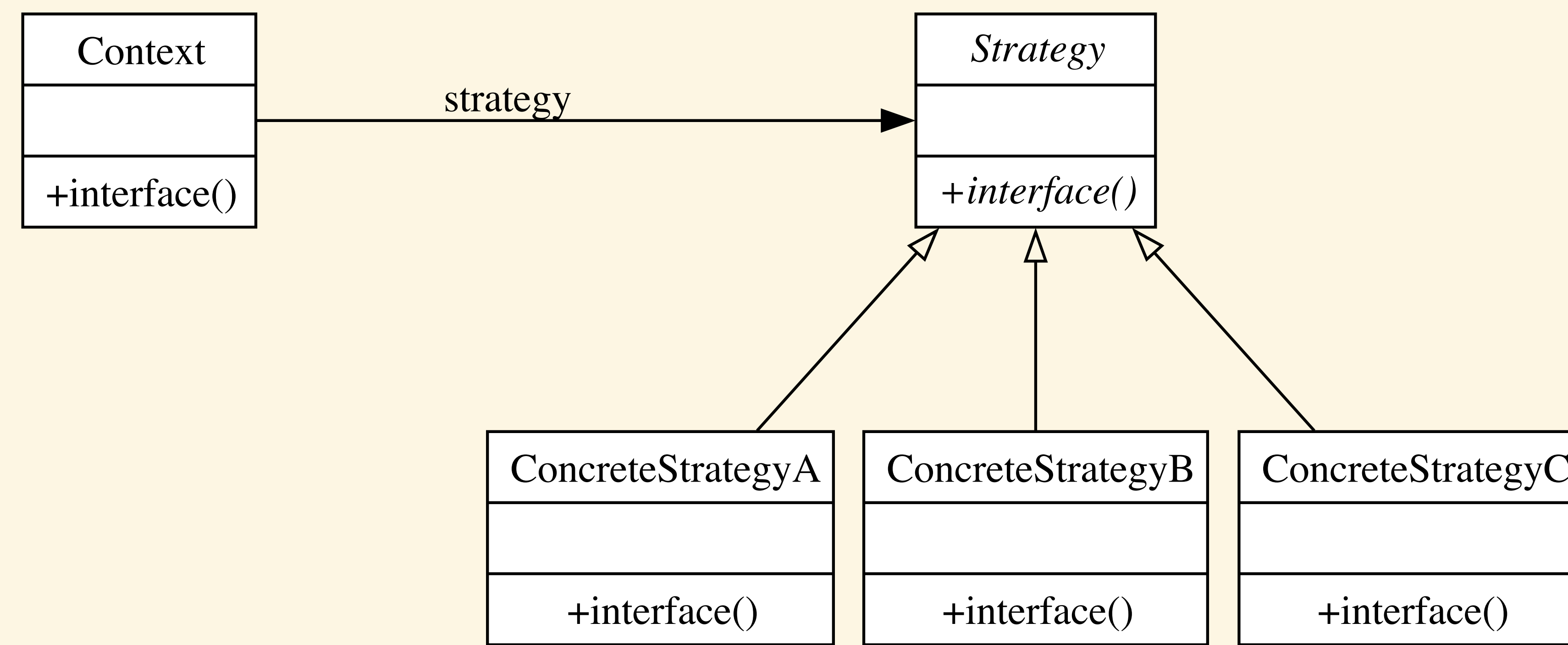


Strategy: Applicability



- Many related classes differ only in behavior
- Different variants of an algorithm are needed, often for different space/time tradeoffs
- The algorithm uses data the client should not know about or had dependencies we want to leave out of the client
- A class has many behaviors, and the operations have multiple conditional statements. Move the switch statement into its own Strategy class

Strategy: Participants



- Strategy (e.g., `Compositor`)

  Declares an interface common to all supported algorithms

  Context uses this interface to call the algorithm defined by a ConcreteStrategy

- ConcreteStrategy (e.g., `SimpleCompositor`, `TeXCompositor`, `ArrayCompositor`)

  Implements the algorithm using the Strategy interface

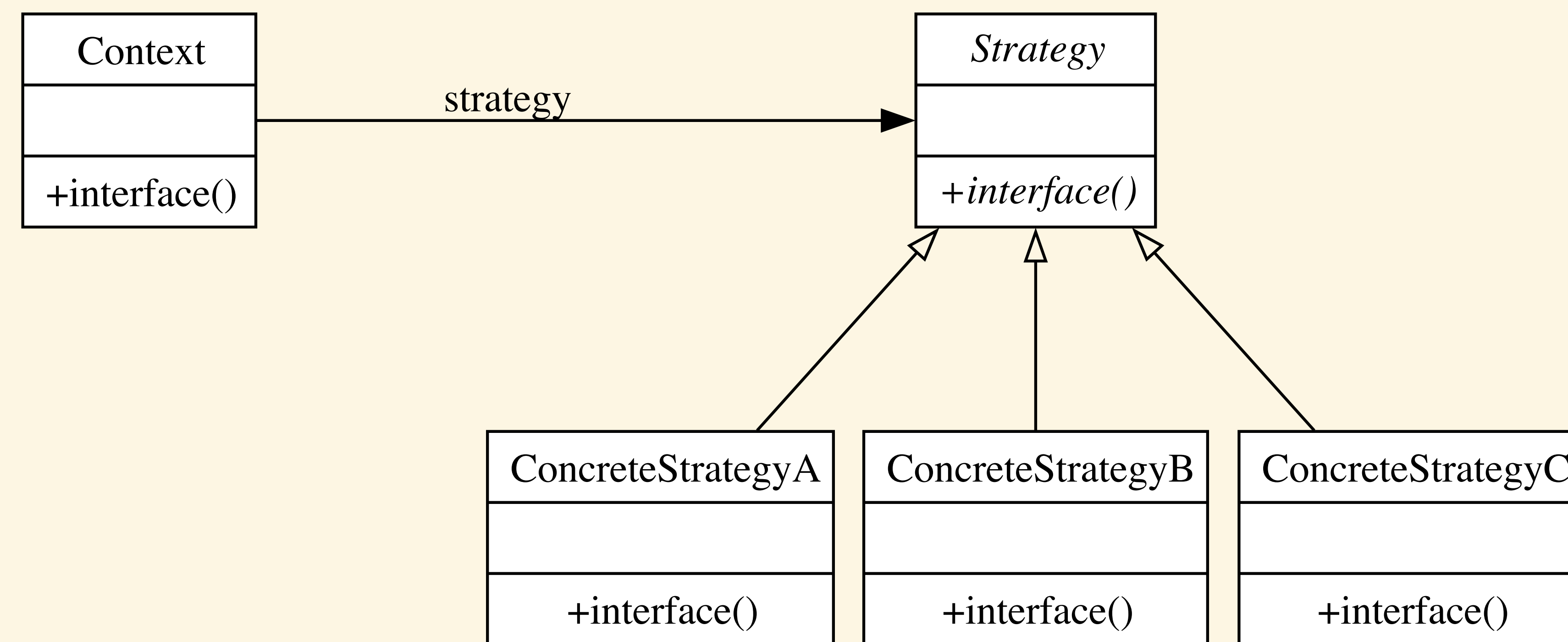
- Context (e.g., `Composition`)

Con
obje

Mai
obje

May
Strat

Strategy: Collaborations



- *Strategy* and *Context* interact to implement the chosen algorithm:

  A *Context* passes data required by the algorithm to the *Strategy* in the call

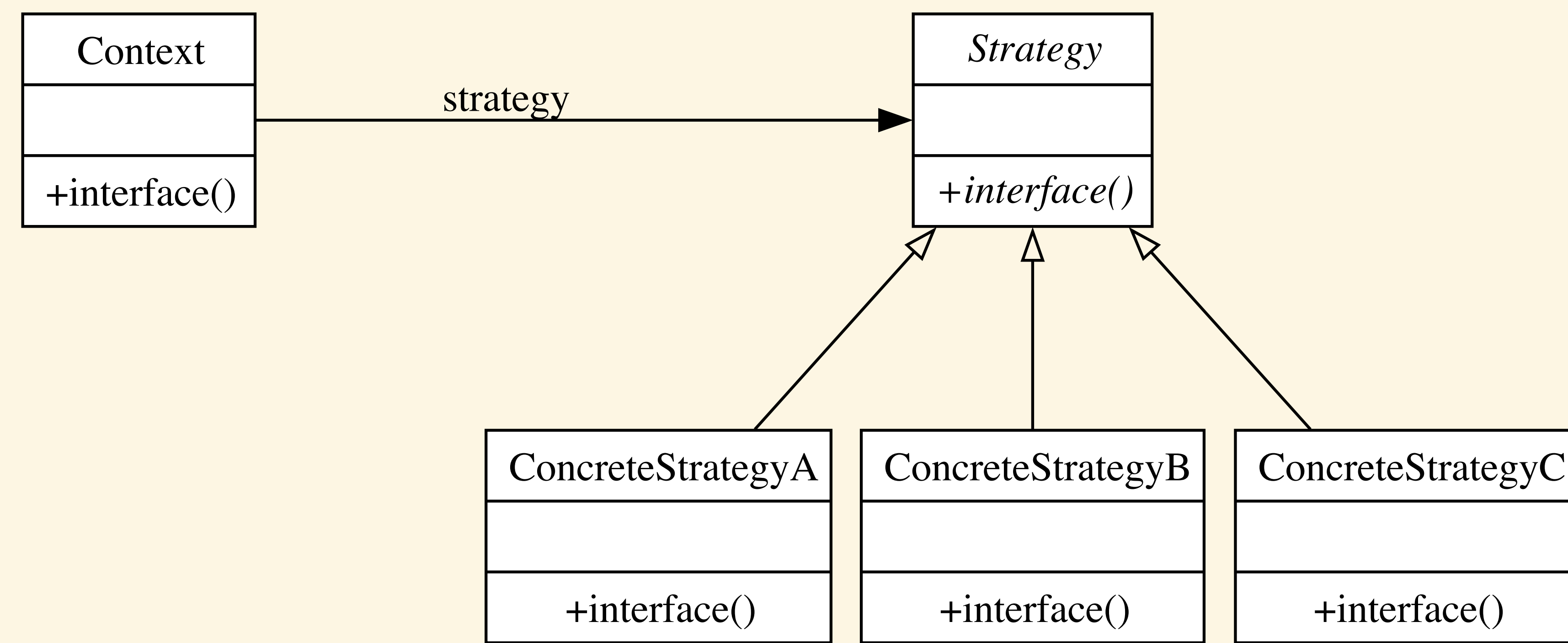
  A *Context* can pass itself as an argument to *Strategy* operations

- A *Context* forwards requests from its clients to its *Strategy*

  Clients usually create and pass a ConcreteStrategy to the context

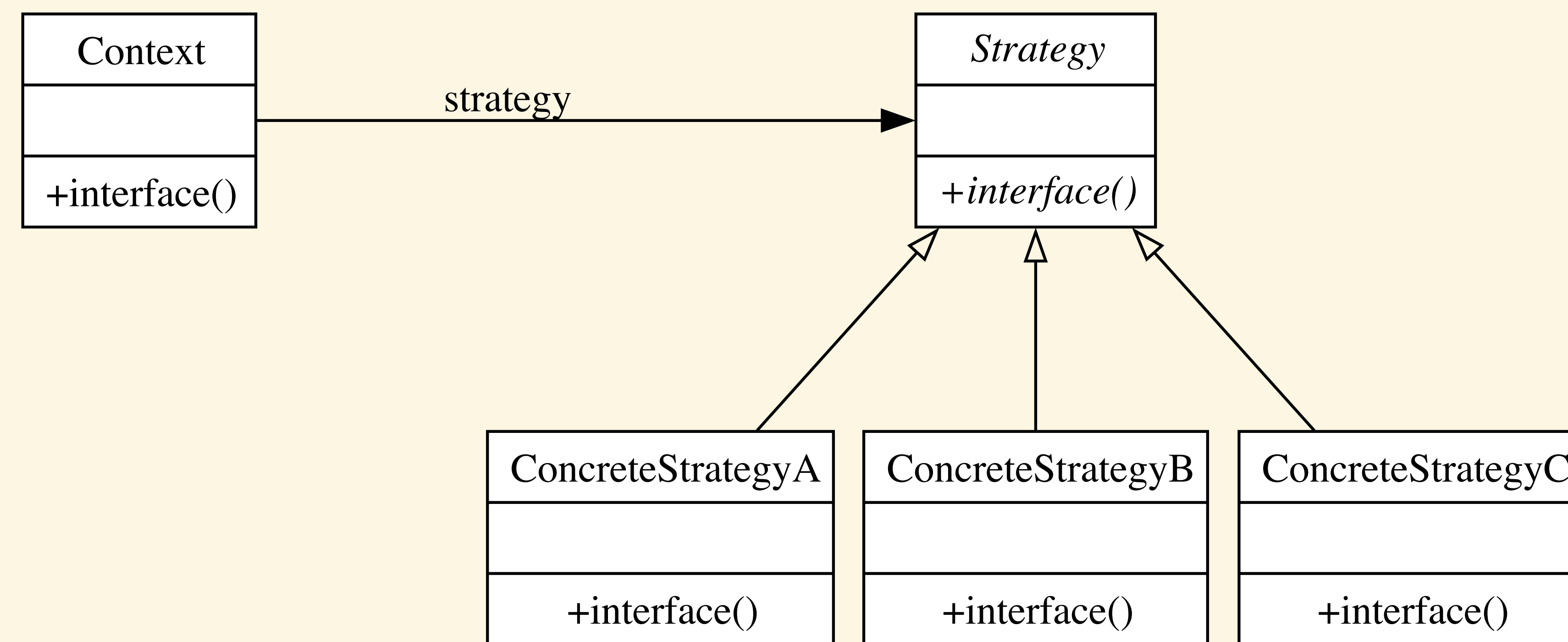
  Typically, the client chooses from a family of ConcreteStrategy classes

Strategy: Advantages



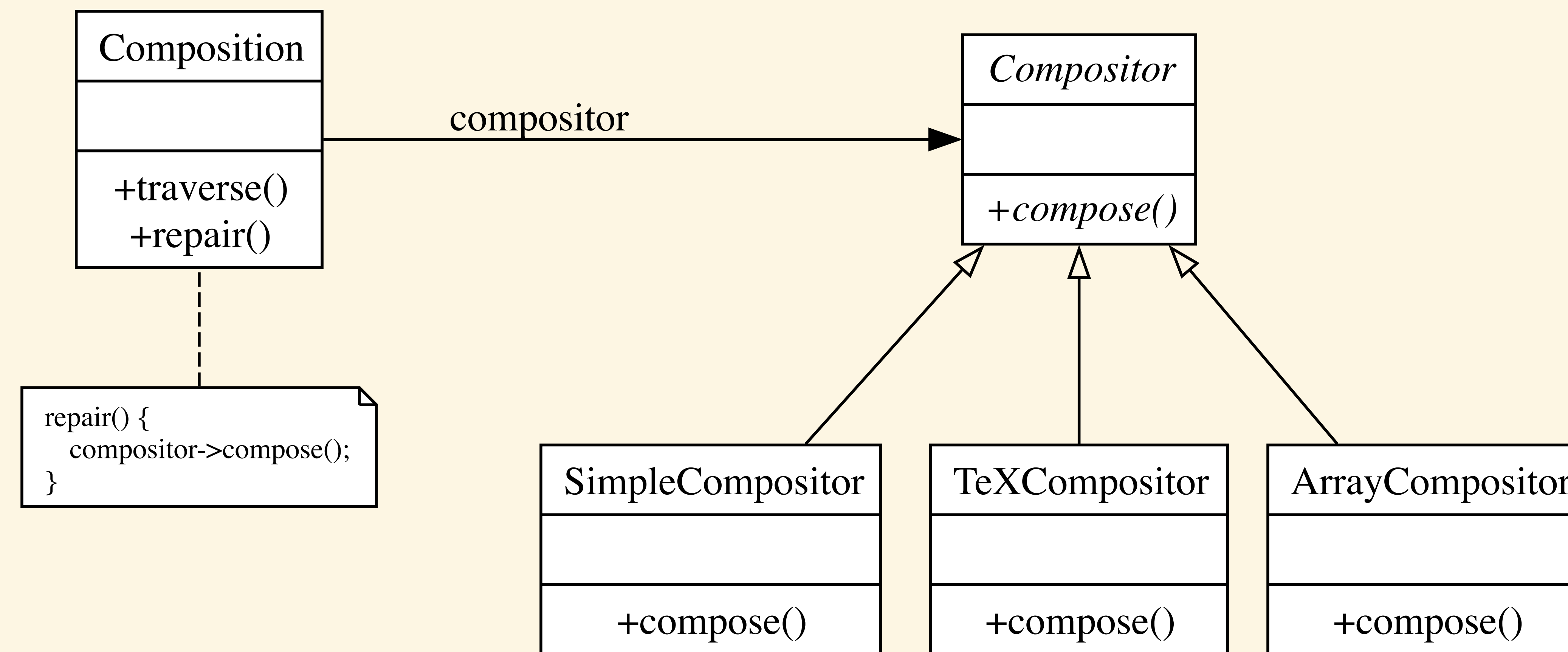
- Families of related algorithms
- An alternative to subclassing the *Context* class
- Eliminate conditional statements
- Wide choice of implementation

Strategy: Disadvantages



- Clients must pick which algorithm to use
- Communication overhead between Strategy and Context
- Increases the number of objects

Implementation

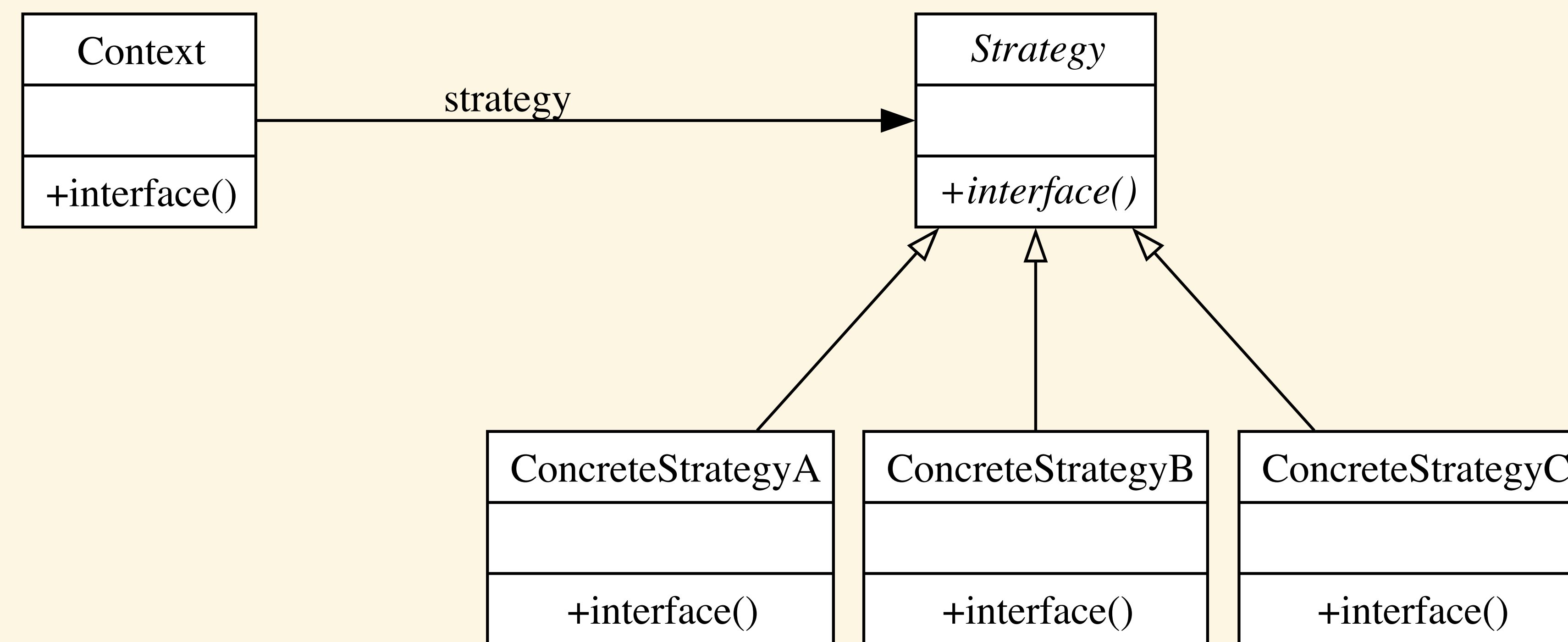



```cpp
void Composition::repair() {
    // ...

    compositor->compose();

    // ...
}
```

Known Uses

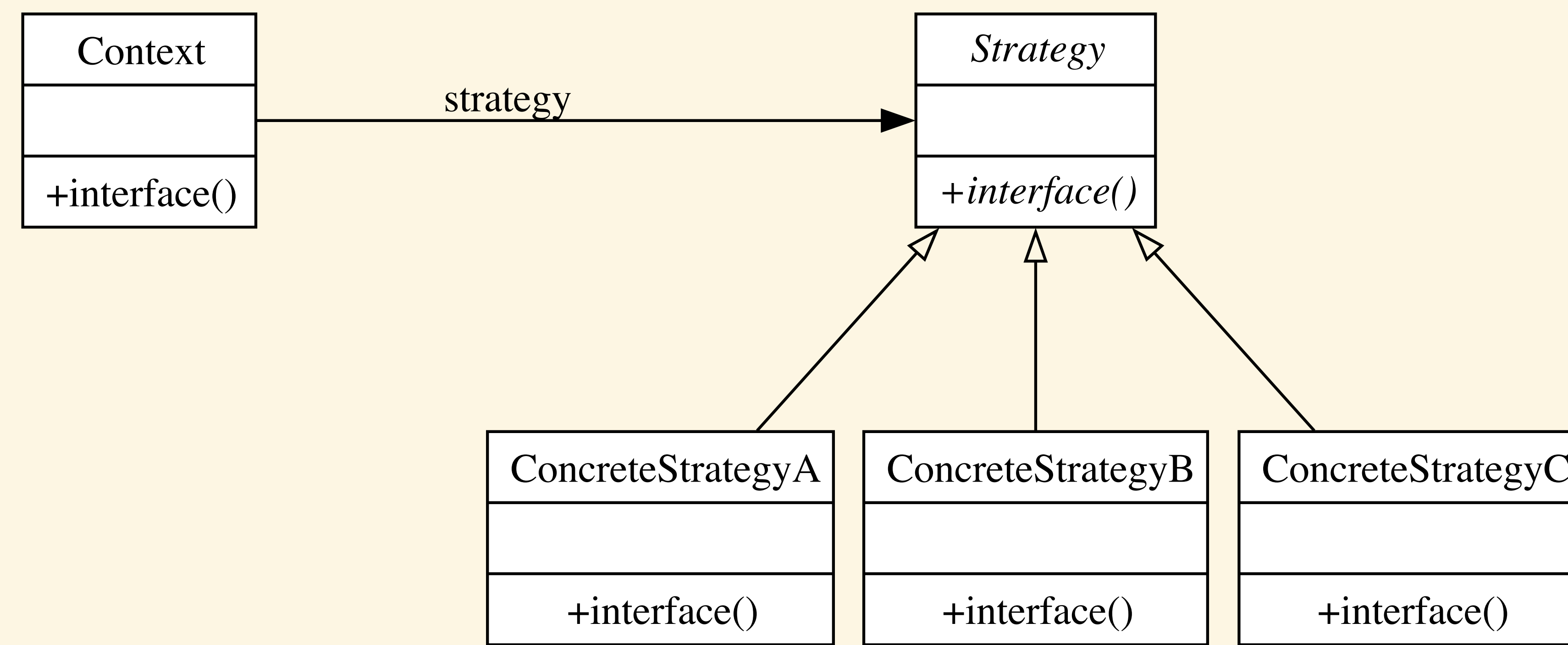


- RTL System for compiler code optimization [JML92]- strategies define different register allocation schemes (RegisterAllocator), and instruction set scheduling policies (RISCscheduler, CISCscheduler)

- ET++SwapsManager calculation engine framework computes prices for different financial instruments [EG92]

  ConcreteStrategy classes for generating cash flows, valuing swaps, and calculating discount factors

Related Patterns



- *Template Method*
- *Flyweight*