CS 247: Software Engineering Principles

Design Patterns

Reading: Freeman, Robson, Bates, Sierra, Head First Design Patterns, O'Reilly Media, Inc. 2004

Ch 1 Strategy Pattern

Ch 7 Adapter and Facade patterns

Ch 8 Template Method

Today's Agenda

Design patterns: codified solutions that put design principles into practice, to improve the modularity of our code.

OO Basi

Separation of Concern

Loose Coupling

Information Hiding

Avoid duplicate code

Encapsulate what is likely to change

OO Principl

Program to an Interface, not a

Liskov Substitutability Principle

Favour Composition over Inheritance

Single Responsibility Principle

Principle of Least Knowledge (Law of Demeter)

Design Patterns

Template Method

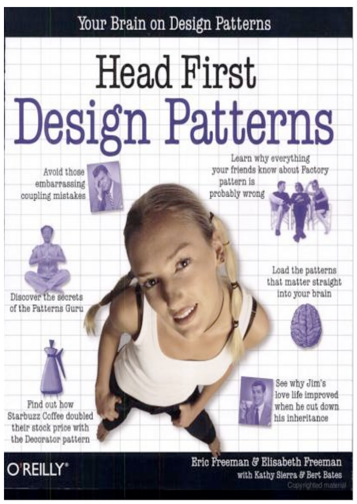
Façade

Adapter

Strategy

References

Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.



Freeman and Freeman, *Head First Design Patterns*, O'Reilly, 2004.

Code examples:

http://headfirstlabs.com/books/hfdp/

Examples from SE_2013:

http://www.student.cs.uwaterloo.ca/~cs247/current/patterns.shtml

What are GoF* Design Patterns?

Abstract OO designs that encapsulate change, to improve the modularity and flexibility of our code.

i.e., increase cohesion, loosen coupling, improve information hiding

How patterns help:

- Improve our efficiency in finding a suitable design
- Improve the predictability of the design quality
- Offer higher-level abstractions than procedures or classes
- Extend developers' vocabulary
- Ease refactoring and evolution

---

*GoF = "Gang of Four", authors of book on object-oriented Design Patterns book

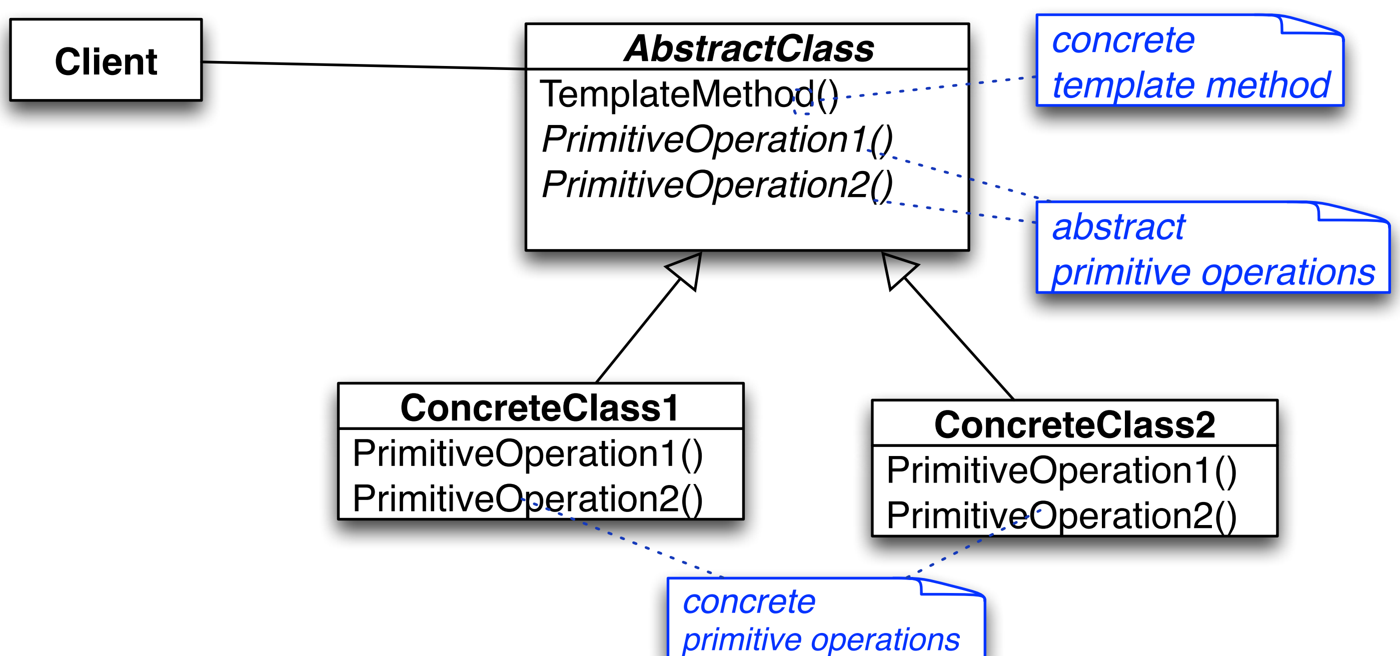
Template Method Pattern

Problem: duplicate code

Multiple subclass methods have similar algorithm structures

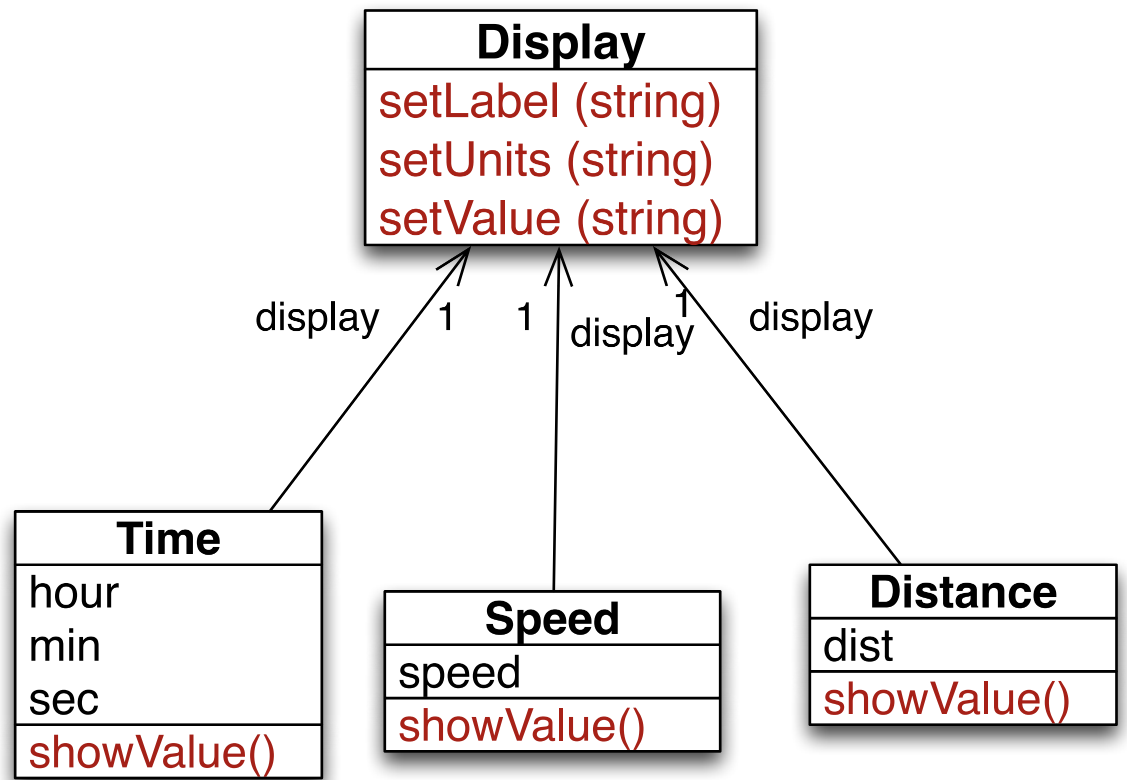
Solution: localize duplicate code structure in an abstract class

Abstract class defines a template method (of common code) that calls pure virtual subroutines. Subclasses override the subroutines



Example: Bicycle Computer

Mode button to toggle among multiple functions to display



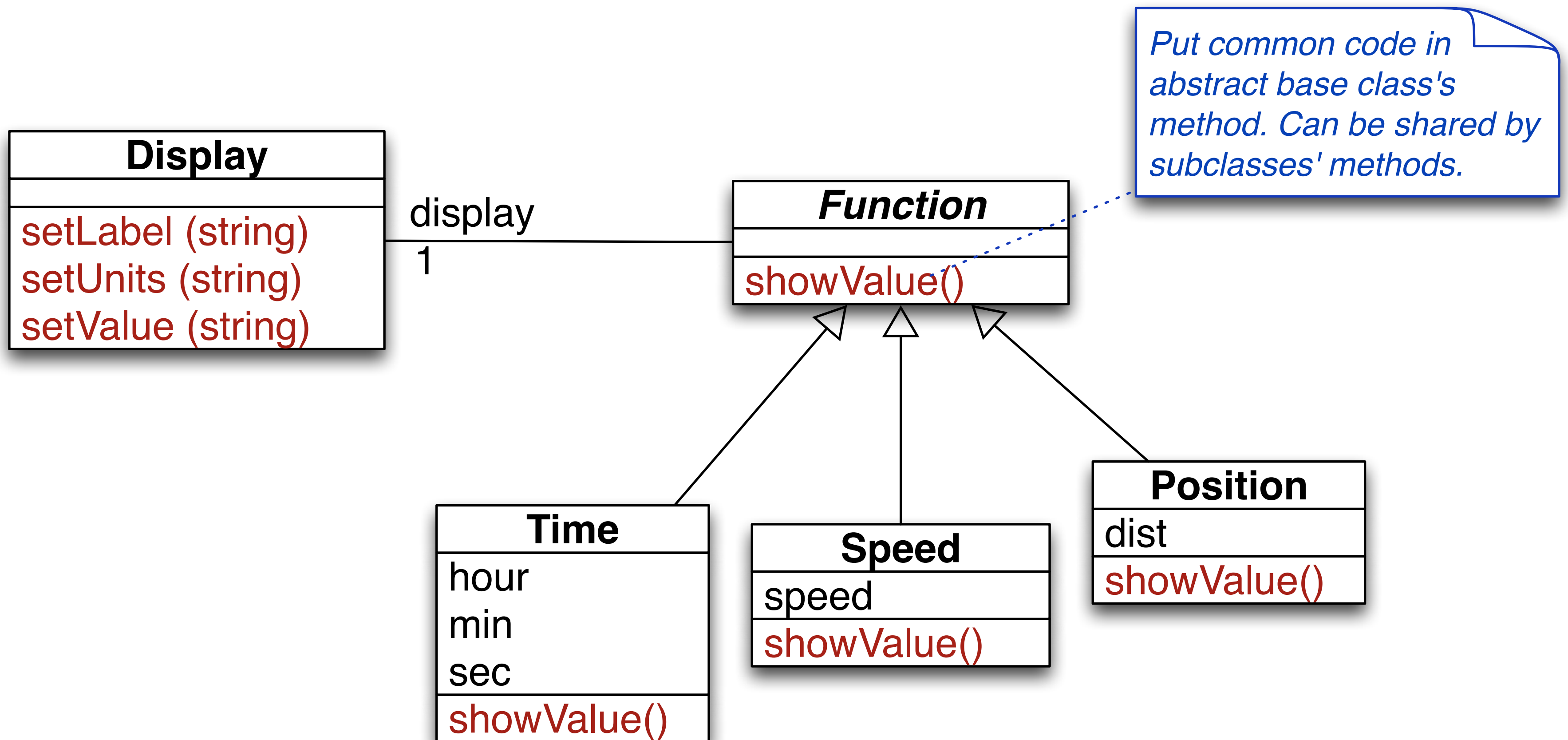
Similar Algorithms

```
void Distance::showValue ()
{
    display_.setLabel ("D");
    display_.setValue (dist_);
    display_.setUnits ("km");
}
```

```
void Time::showValue ()
{
    display_.setLabel ("T");
    display_.setValue (hour_ + ":" + min_ + ":" + sec_);
    display_.setUnits ( "" );
}
```

```
void Speed::showValue ()
{
    display_.setLabel ("S");
    display_.setValue (speed_);
    display_.setUnits ("km/h");
}
```

Approach 1: Inheritance



Downside: subclasses may neglect to use parent behaviour.

Inheriting Common Code

```
void Function::showValue ()
{
    display_.setLabel (XXX);
    display_.setValue (YYY);
    display_.setUnits (ZZZ);
}
```

Revisit: The commonality among the classes' showValue() methods is the structure of the algorithm.

The variations are data values (but the data values could just as easily be subroutine values).

Approach 2: Template Method

A template method is a base-class method that defines common code structure, but includes primitive operations (holes) to be defined by subclass methods.

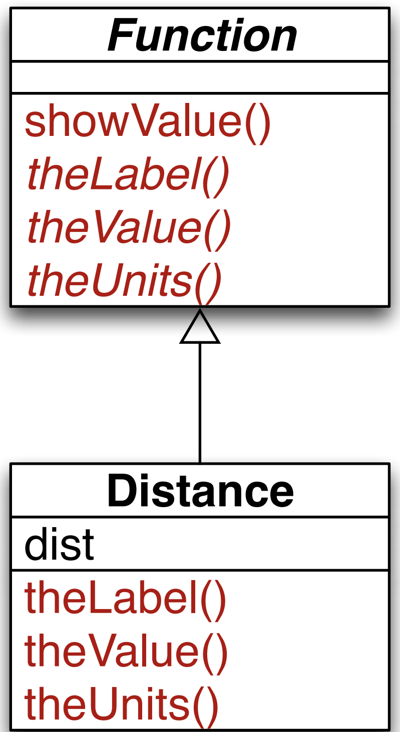
*Better that derived classes override **holes** than override **entire method***

```
void Function::showValue (){
    display_.setLabel ( theLabel() );
    display_.setValue ( theValue() );
    display_.setUnits ( theUnits() );
}

// pure virtual methods

std::string Function::theLabel () = 0;

std::string Function::theValue () = 0;

std::string Function::theUnits () = 0;
```

It is essential that

- the template method be nonvirtual
- the primitive operations be virtual functions in the base class

Template Method



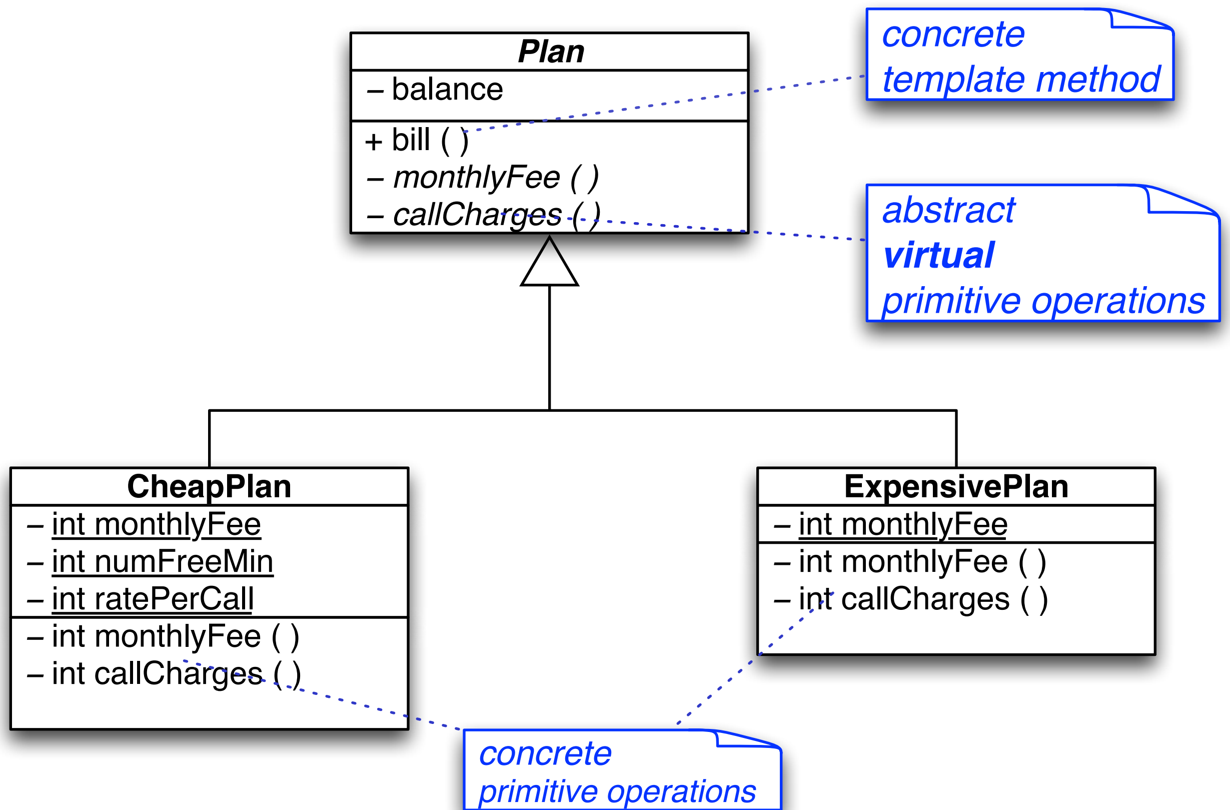

```cpp
// template method
void Function::showValue () {
    display_.setLabel ( theLabel() );
    display_.setValue ( theValue() );
    display_.setUnits ( theUnits() );
}

// primitive operations
std::string Function::theLabel() = 0;
std::string Function::theValue() = 0;
std::string Function::theUnits() = 0;
```

```cpp
std::string Distance::theLabel() {
   return string("D");
}
std::string Distance::theValue() {
   return string(dist_);
}
std::string Distance::theUnits() {
   return string("km");
}
```

Example: Assignment 1

Operation that bills a cellphone plan subscriber, based on plan type

- monthly fee
- charges for additional calls

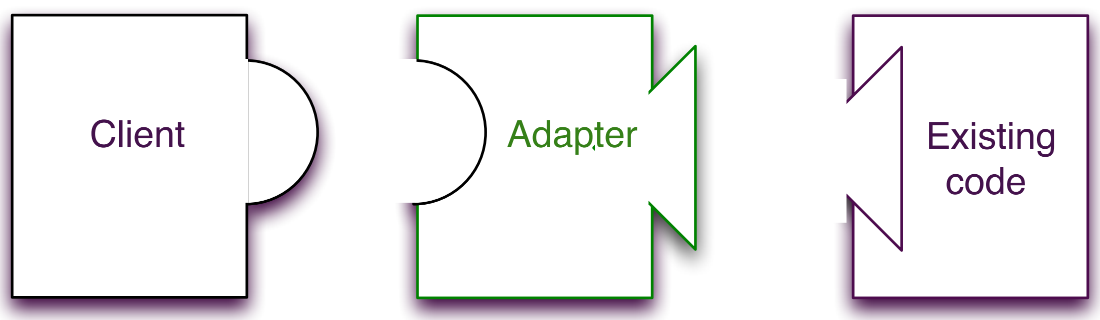


Adapter Pattern Idea

Problem: Interface mismatch between two modules

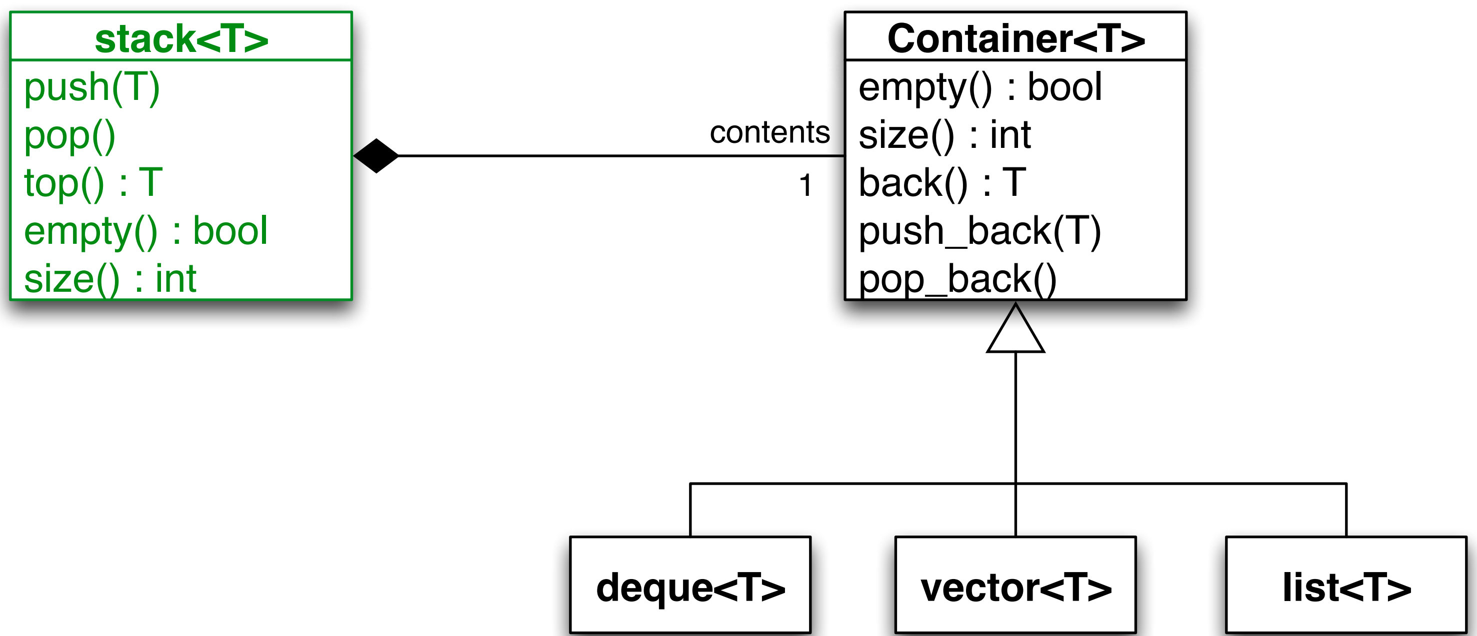
- Want to reuse an existing class, but its interface does not match what is needed
- Or the interface of one of our modules changes (!!) and we don't want to make major changes to the existing (working!) code.

Solution: Define an Adaptor class that maps one interface to another



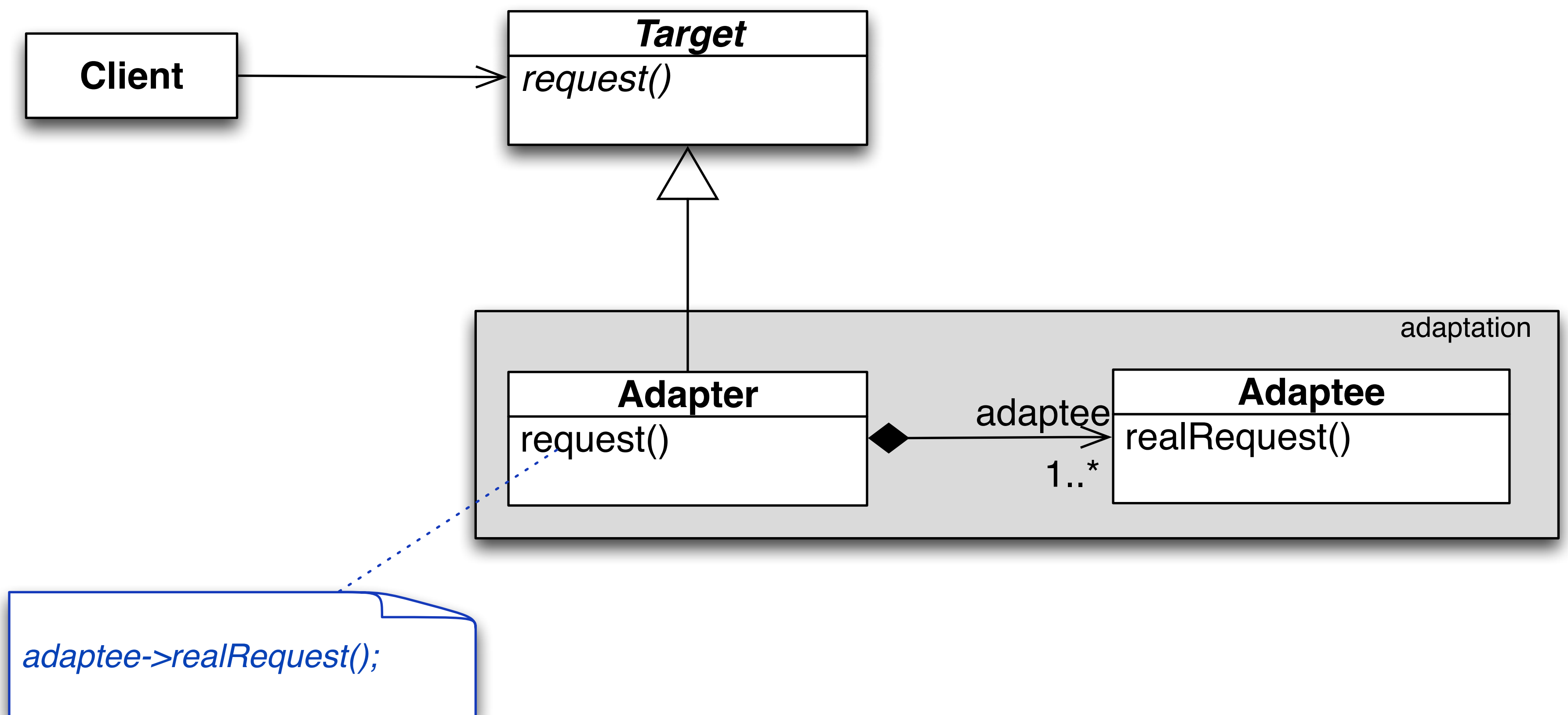
Example: STL Stacks

STL stacks are implemented by adapting the interface of a Container class (by default, deque).



Adapter Pattern
(object adapters)

The Adapter class translates requests made to the Target interface into requests made of the Adaptee object.



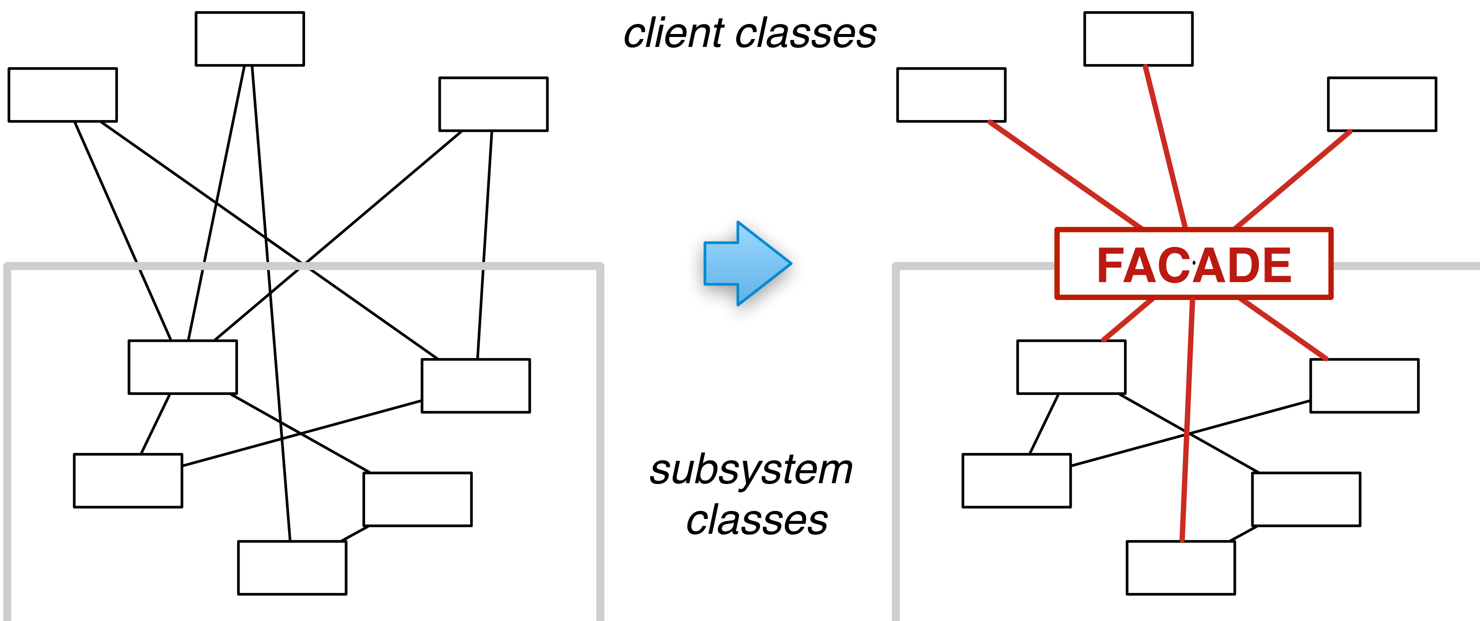
Facade Pattern

Problem: complex interface

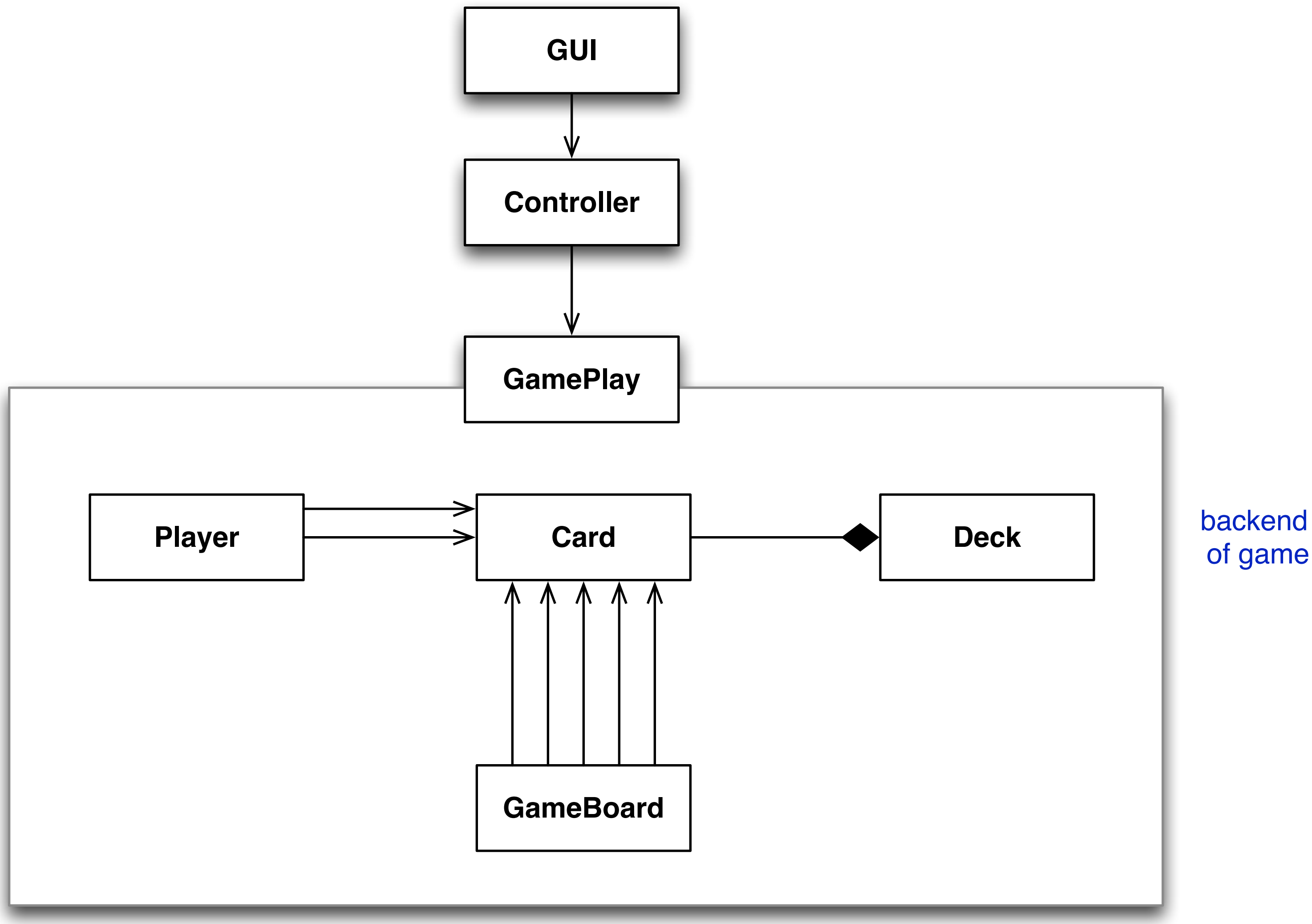
Client of subsystem interacts with multiple (complex?) classes

Solution: create a single, simplified interface (class)

Restrict, simplify client's interactions with subsystem's classes



Example: Project



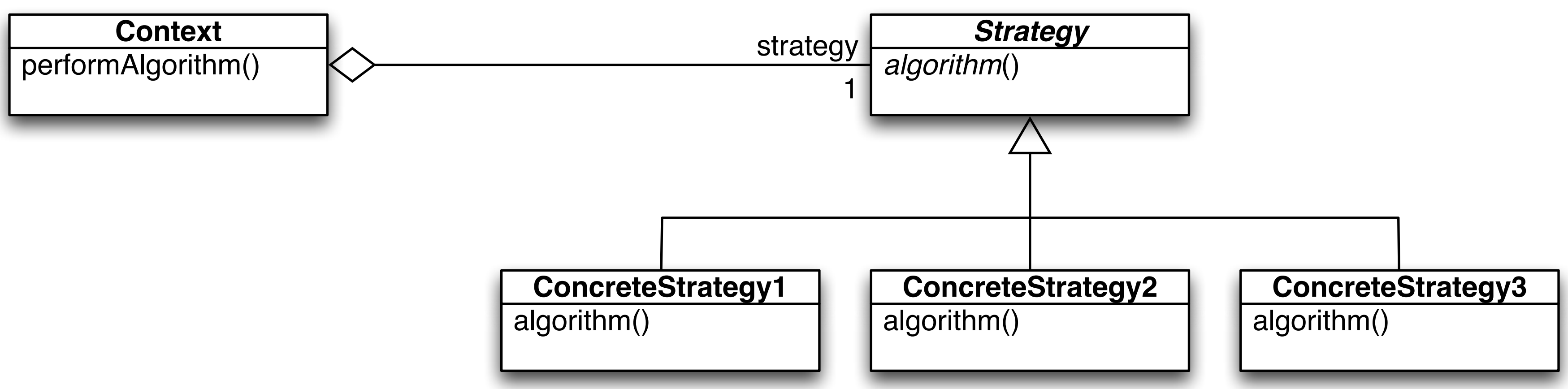
Strategy Pattern

Problem: Want to vary an algorithm at run-time

Solution: Encapsulate the algorithm decision

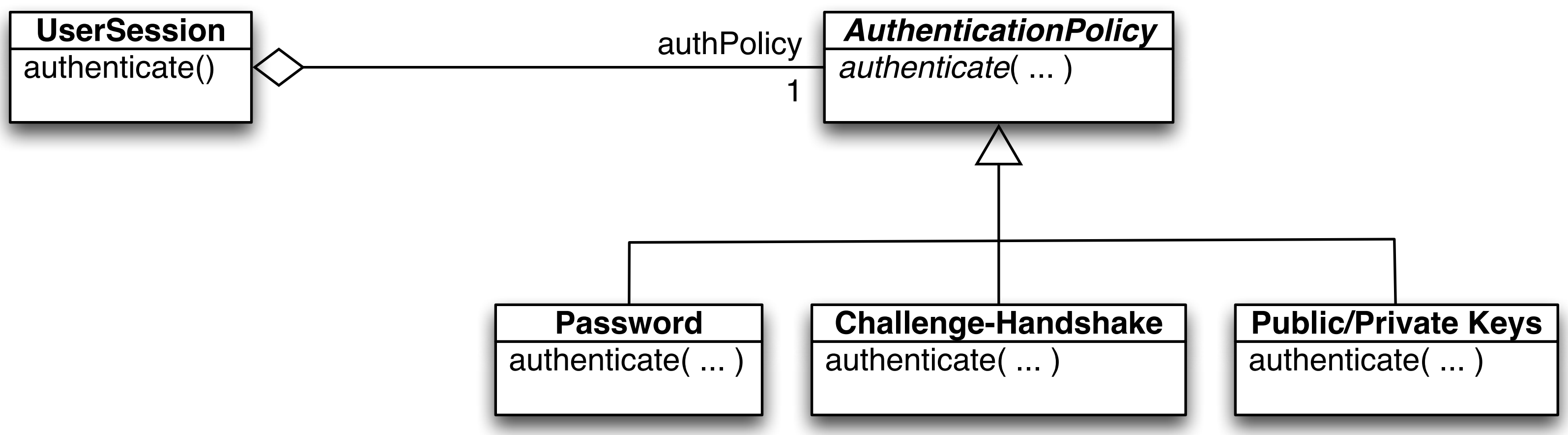
Define algorithm as a component object

Use subclassing to specialize the algorithm in different ways



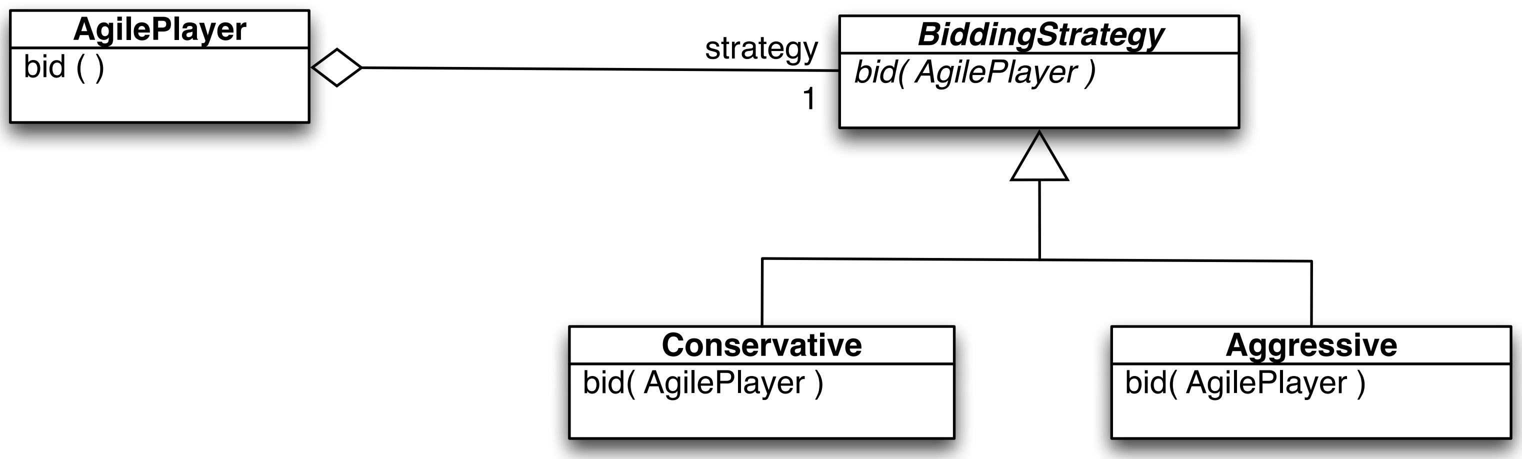
Example 1

A class that authenticates users, and uses a variety of authentication policies.



Example 2

A card player whose strategy changes at runtime.



Considerations

Applicability:

- Different variants of an algorithm
- Choose/Change algorithm at runtime

Some complications:

- Encapsulation of context data to be operated on
- All algorithms must use the same Strategy interface (e.g., sort)

Summary

The goal of design patterns is to encapsulate change

Template Method Pattern: encapsulates that parts of an algorithm that is different for each derived class

Adapter Pattern: encapsulates the interface to a class

Facade Pattern: encapsulates the client's interface to a collection of classes

Strategy Pattern: encapsulates the algorithm to be used, such that it can be set (and changed) at runtime