CS 247: Software Engineering Principles

Object-Oriented Design Principles

Reading: none

Today's Agenda

Object-Oriented Design Principles - characteristics, properties, and advice for making decisions that improve the modularity of the design.

OO Ba

Separation of Concerns
Encapsulate what is lik
Encapsulate Data Repr
Abstraction (interfaces,
Reuse (through compo
Polymorphism

OO Principles

Program to an Interface, not an Implementation

- aka Open Closed Principle

Favour Composition over Inheritance

Single Responsibility Principle

Dependency Inversion Principle

Liskov Substitutability Principle

Principle of Least Knowledge (Law of Demeter)

References

If you want to know more

- Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- Bertrand Meyer, *Object-Oriented Software Construction,* Prentice Hall, 1997
- Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices,* Prentice Hall, 2003.
- Barbara Liskov and John Guttag, *Program Development in Java,* Addison-Wesley, 2000.
- Stanley B. Lippman, *Essential C++*, Addison-Wesley, 2000.

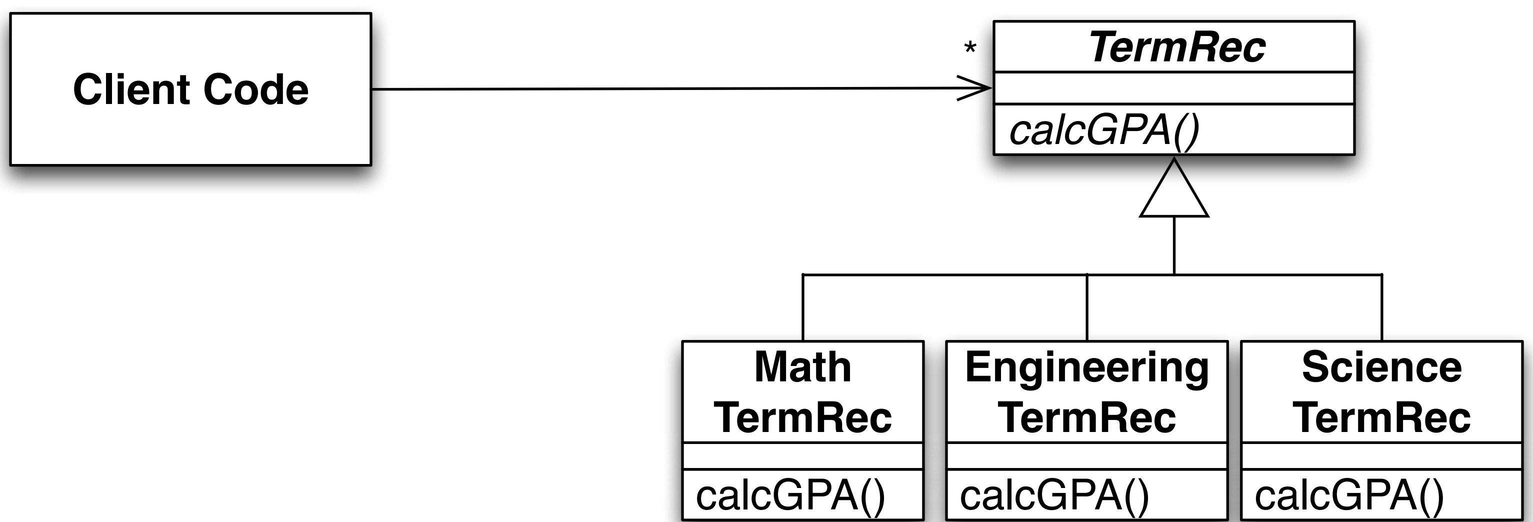
Open Closed Principle

Principle: A module should be open for extension but closed to modification.*

*AKA Program to an Interface, not an Implementation

Idea: Have client code depend on an abstract class (that can be extended) rather than on concrete classes.

Example: Dynamic polymorphism



Inheriting Interface vs. Implementation

The abstract base class designer determines what parts of a member function the derived class inherits:

1. interface (declaration) of member function
2. interface and (default) overridable implementation
3. interface and non-overridable implementation

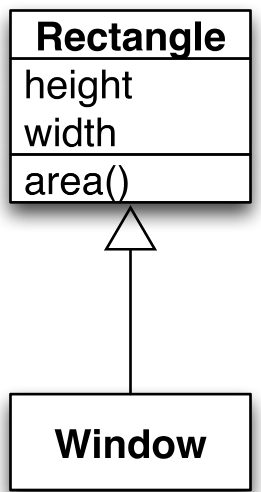
```
class TermRecord {
public:
   virtual void printStats() const = 0; {//template method op}
   virtual float calcGPA () { //compute mean average}
   void print() const; { //print template transcript}
   ...
};

class MathTermRecord : public TermRecord { ... }

class EngineeringTermRecord : public TermRecord { ... }
```

Inheritance vs. Composition

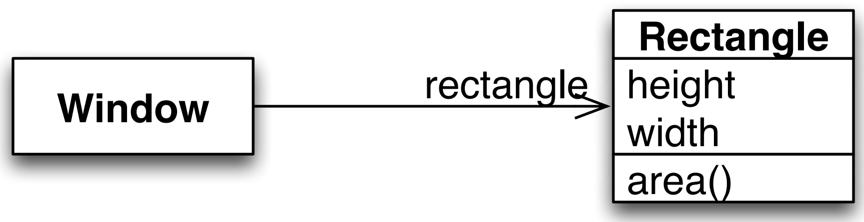
Bertrand Meyer, *Object-Oriented Software Construction*

Problem: When defining a new class that includes attributes and capabilities of an existing class, should our new class

- inherit from the existing class (inheritance)?
- include existing class as a complex attribute (composition)?



"is-a"



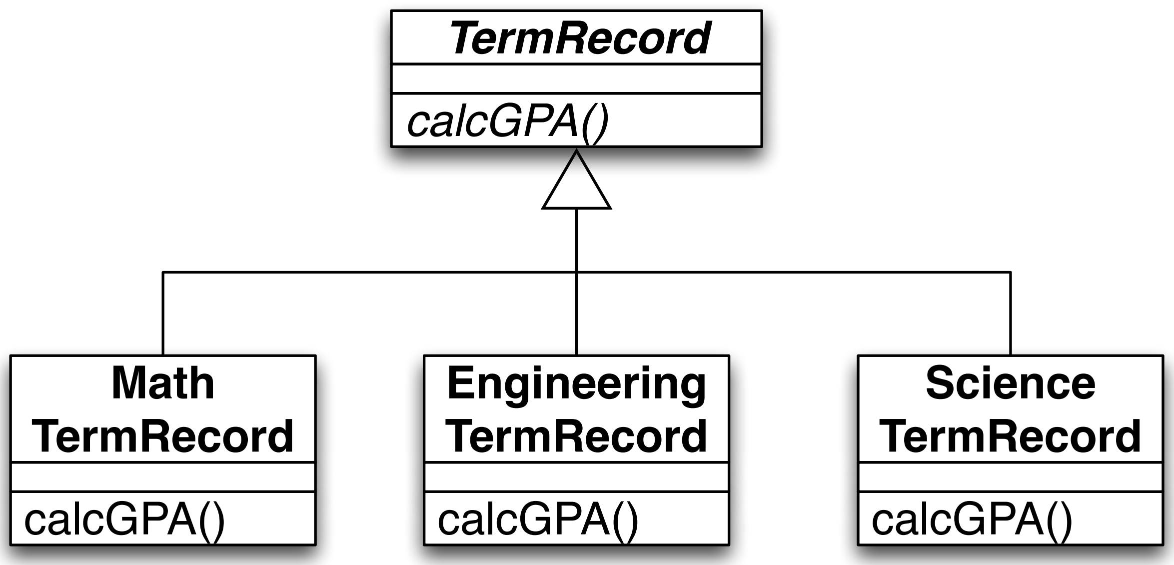
"has-a"

Choosing Inheritance

Principle: Favour inheritance when

(1) using subtyping -- i.e., using the fact that a derived class can be used wherever the base class is accepted

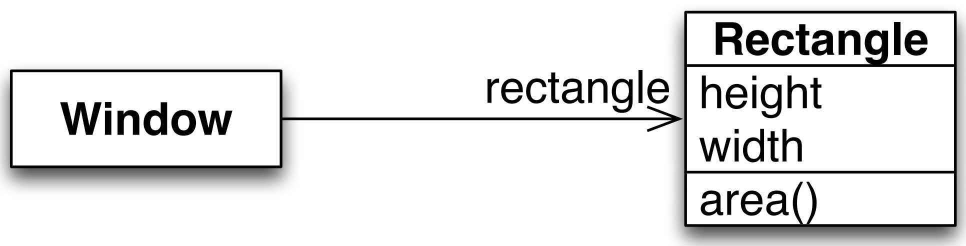
(2) using the entire interface of an existing class



Choosing Composition

Principle: Favour composition for simple (non-overriding) code reuse and extension...

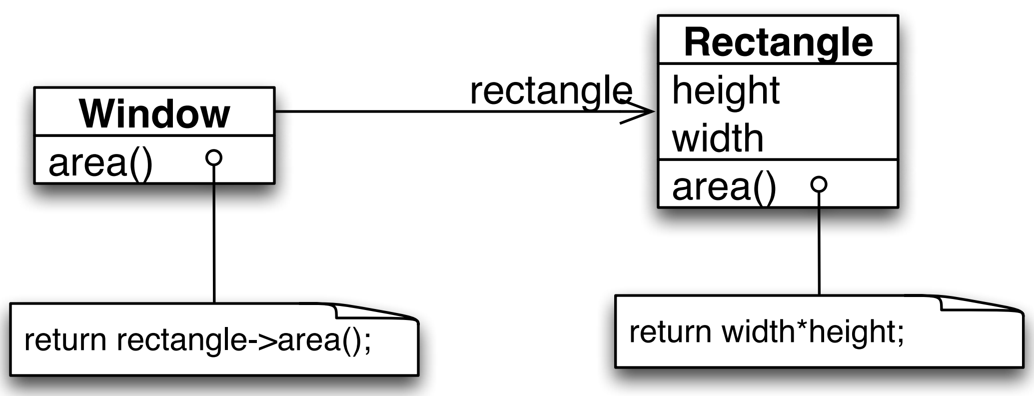
...because with composition, the components' capabilities (data and functions) can change at run-time.



Delegation

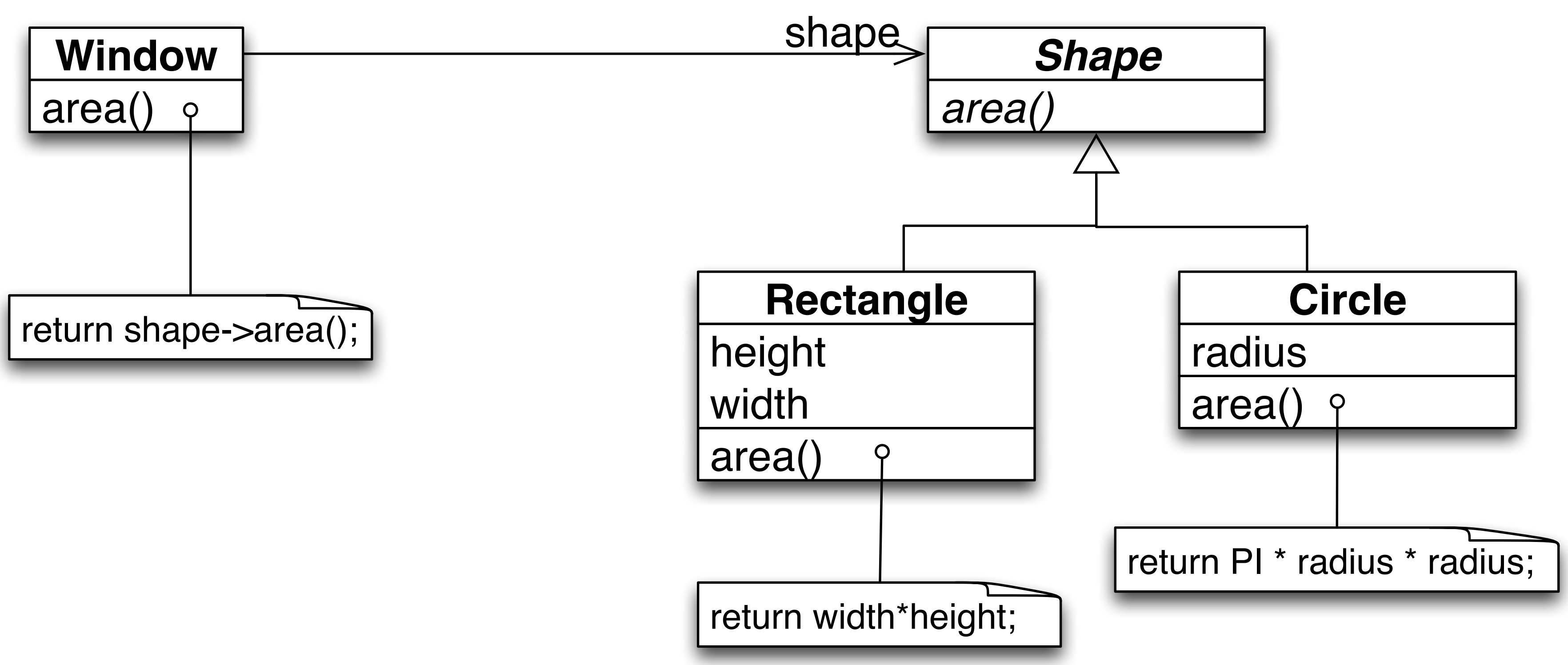
Delegation in object composition simulates inheritance-based method reuse.

- Composite object delegates operations to component object
- Can pass itself as parameter, to let delegated operation refer to composite object



Composition and Open Close Principle

The benefits of composition are maximized when the component is an abstract type — an interface (in Java) or an abstract base class (in C++) — that can be concretized in different ways.

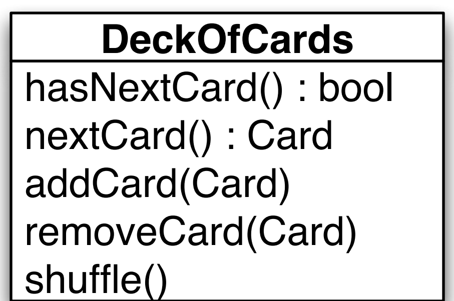


The Single-Responsibility Principle

Principle: Encapsulate each changeable design decision in a separate module.

The Single-Responsibility Principle offers guidance on how to decompose our program into cohesive modules.

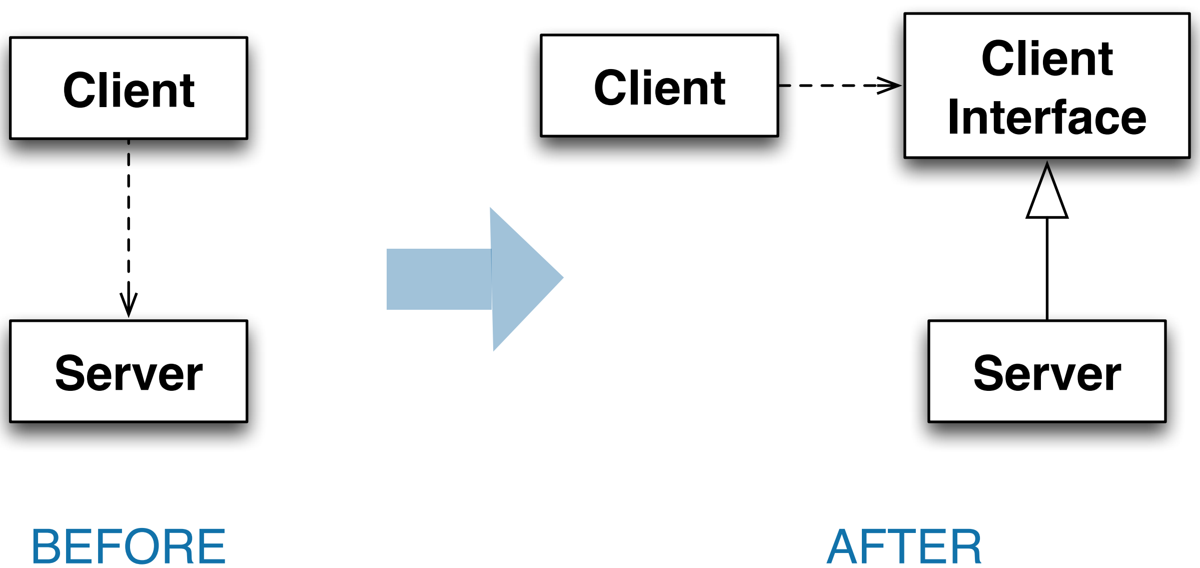
Example: Based on the names of its methods, how many design decisions are implemented by this one class?



Dependency Inversion

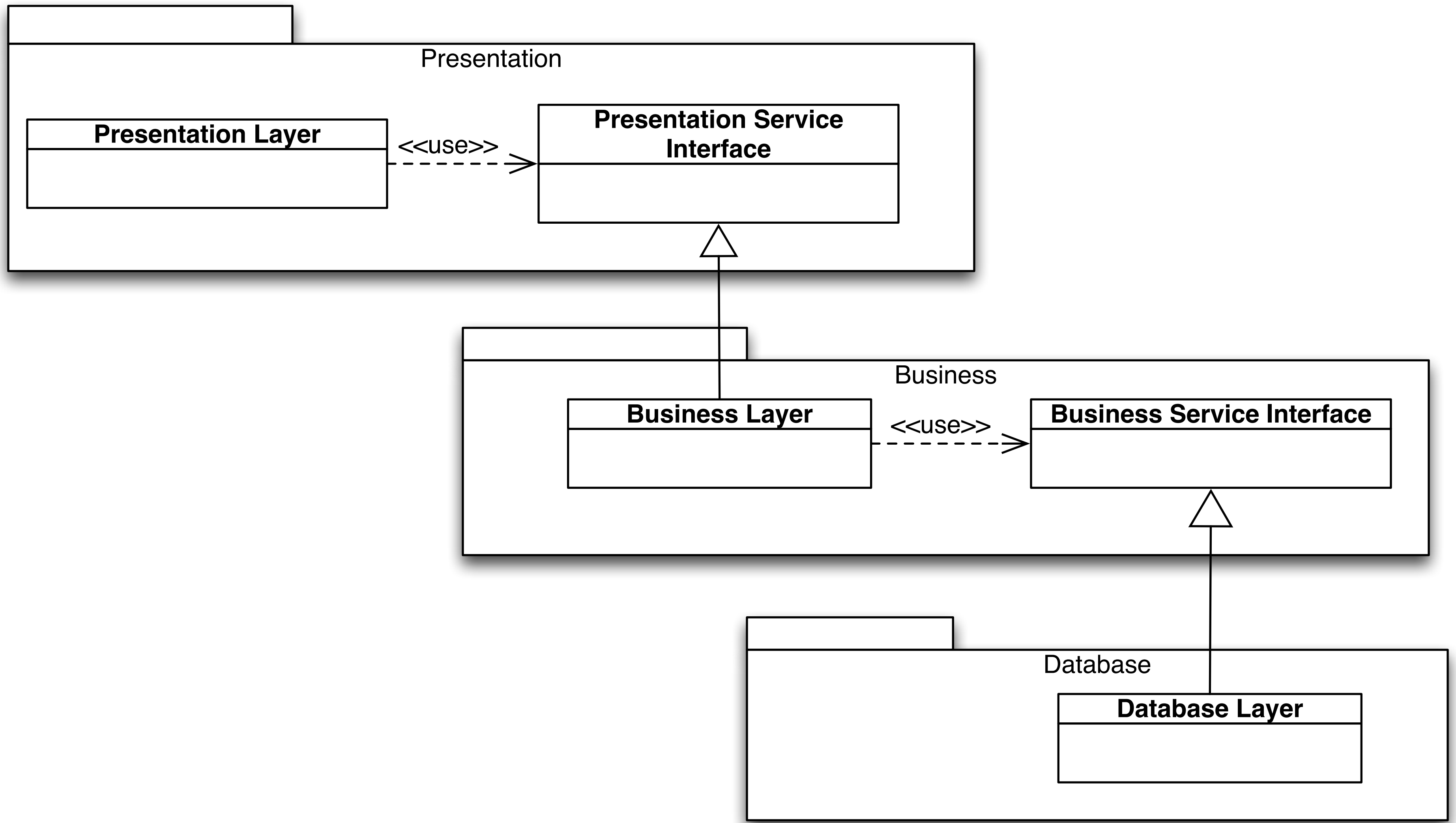
Principle: High-level modules should depend on abstractions rather than on concrete classes.

Idea: "Invert" the normal direction of dependency from client to server module, so that instead the client and server modules both depend on an interface that represents the client's needs.

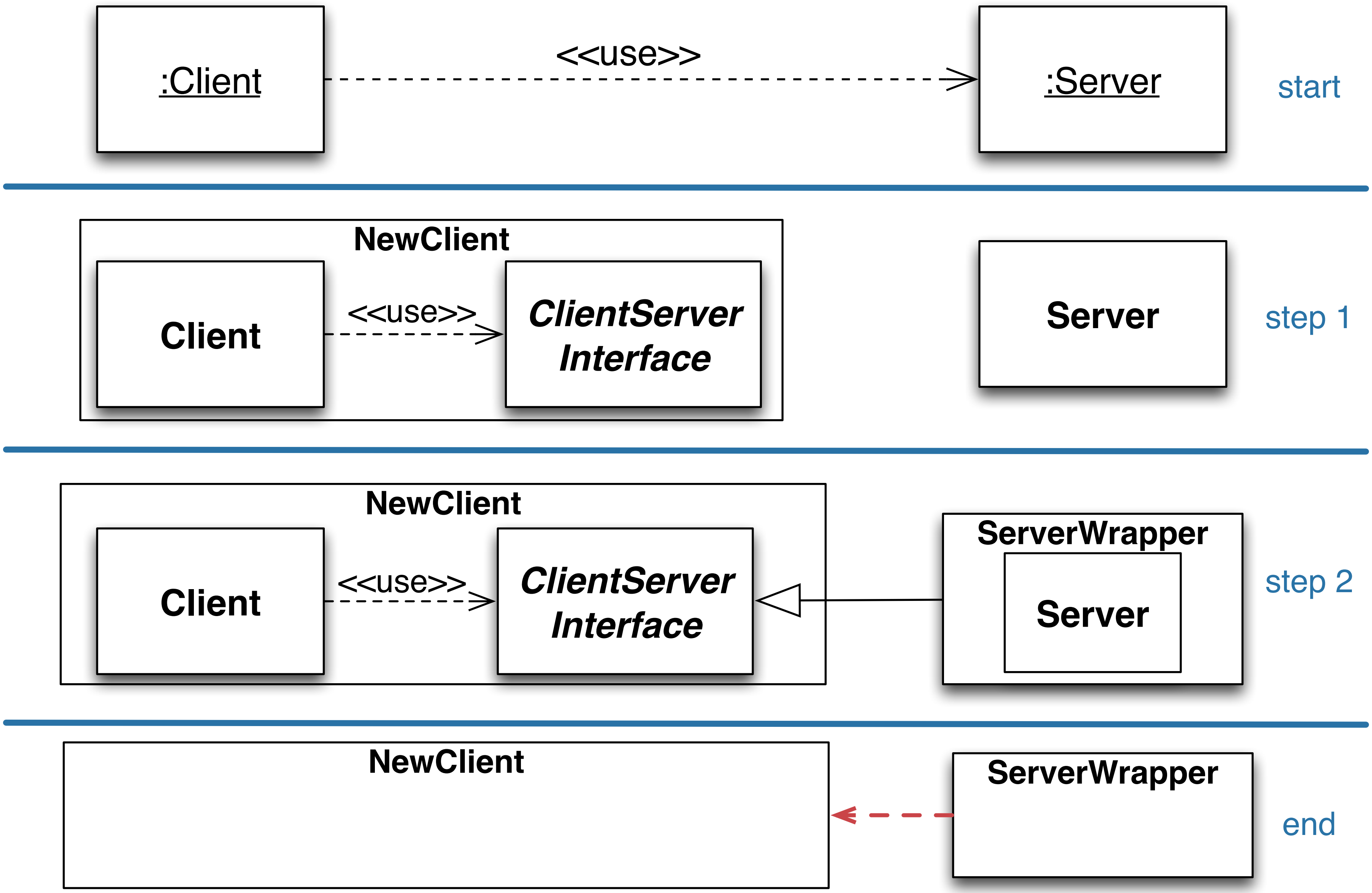


Microsoft Three-Tier Architecture

Microsoft's three-tiered architecture for Web-based applications



Dependency Inversion

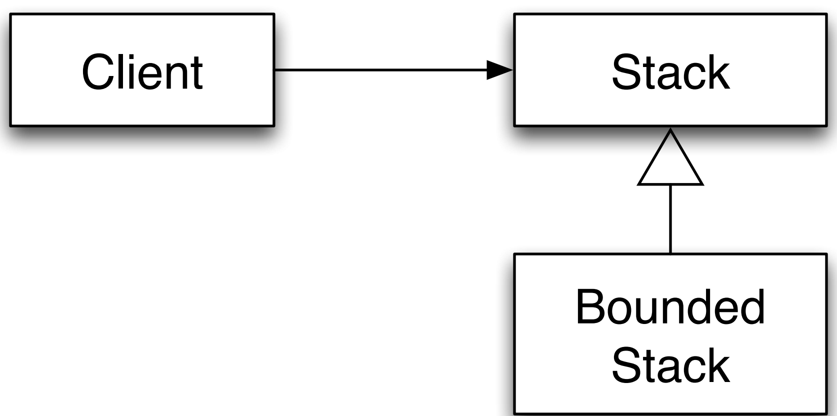


Liskov Substitutability Principle (LSP)

Principle: A derived class must be substitutable for its base class.

Idea: A derived class must preserve the behaviour of its base class, so that it will work with client code that uses the base class.

- objects accept the base class's messages
- methods require no more than base class methods
- methods promise no less than base class methods



Substitutability Rules

Liskov, Guttag, *Program Development in Java*

When overriding inherited virtual functions, three rules must be followed:

1) Signatures: The derived-class objects must have all of the methods of the base class, and their signatures must match.

2) Method behaviours: Calls to derived-class methods must behave like calls to the corresponding base-class methods.

3) Properties: The derived class must preserve all properties of the base class objects.

LSP Signature Rules

Liskov, Guttag, *Program Development in Java*

Signatures: The derived class must support all of the methods of the base class, and their signatures must match:

- Parameters of overridden virtual methods must have compatible types as the parameters of the base class's methods.
  - C++, Java: same types (otherwise redefining)
- The return type of an overridden virtual method must be compatible with the return type of the base-class's method.
  - C++, Java: same type, or subtype of
- A derived-class method raises the same or fewer exceptions than the corresponding base-class method.

LSP Method Rules

Liskov, Guttag, *Program Development in Java*

Method behaviours: A derived-class method maintains or weakens the precondition and maintains or strengthens the postcondition:

- Precondition Rule:
  pre_base => pre_derived

- Postcondition Rule:
  (pre_base && post_derived) => post_base

In other words, the specification of a derived class must be stronger-than the specification of its base type.

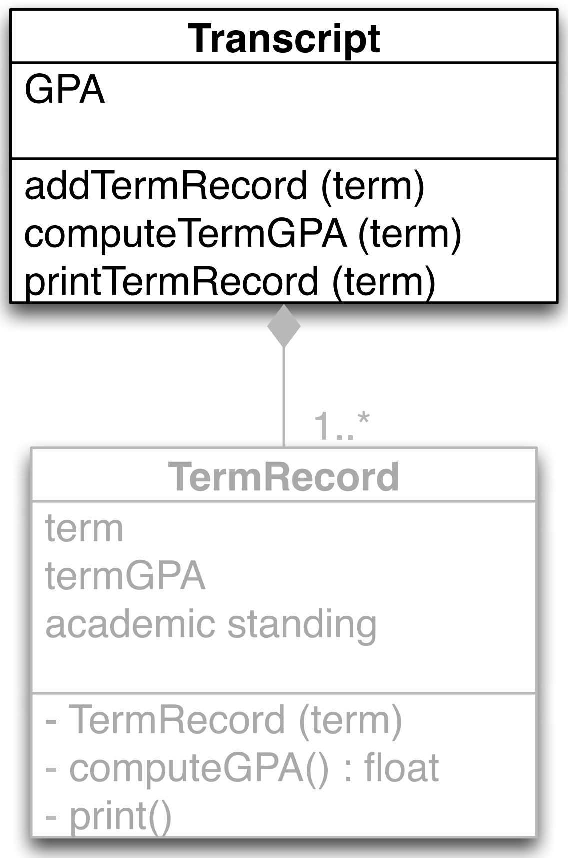
LSP Property Rules

Liskov, Guttag, *Program Development in Java*

Property behaviours: The derived class must preserve all declared (and enforced) properties of the base class objects.

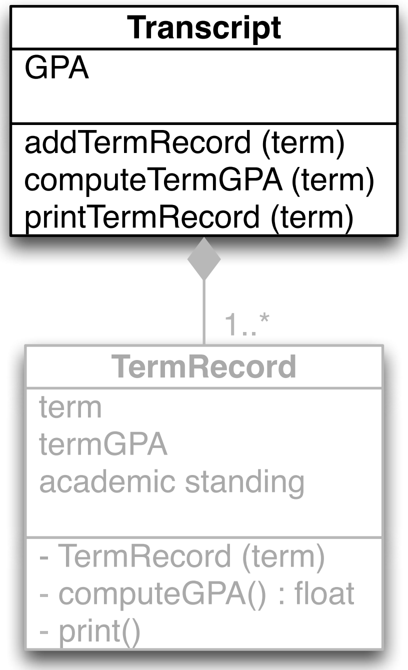
- invariants (e.g., no duplicate elements in a container type)
- optimized for performance (memory requirements, timing)

Encapsulation of Components



Information Hiding: Modular design should hide design and implementation details, including information about components.

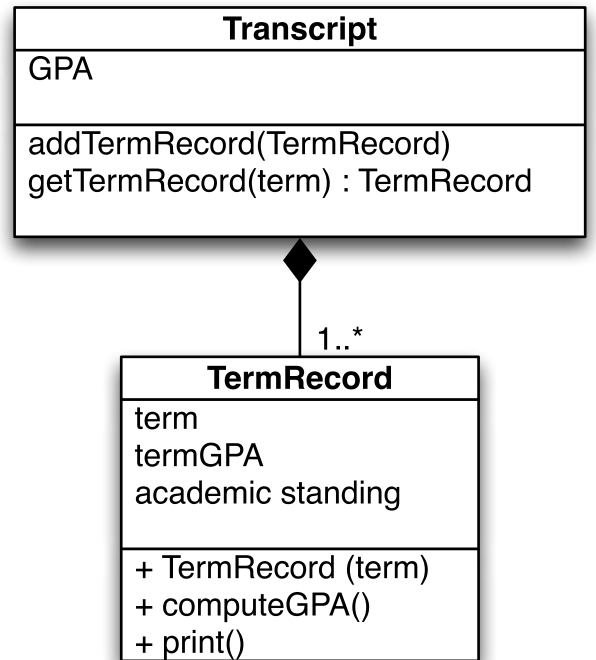
Composition and Data Encapsulation



Client Code

```
Transcript *t;
...
t->addTermRecord("Spring2014");
t->computeTermGPA("Spring2014");
t->printTermRecord("Sring2014");
```

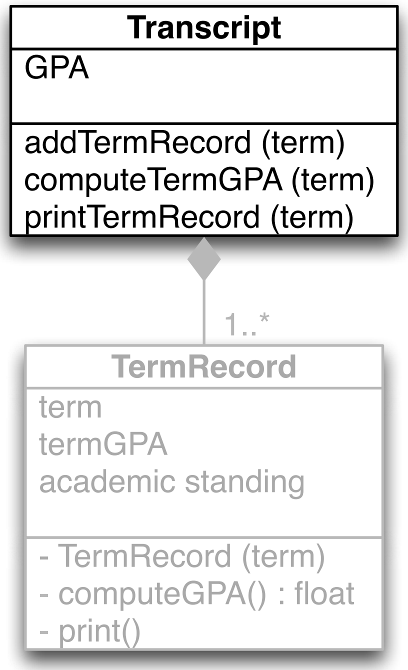
vs.



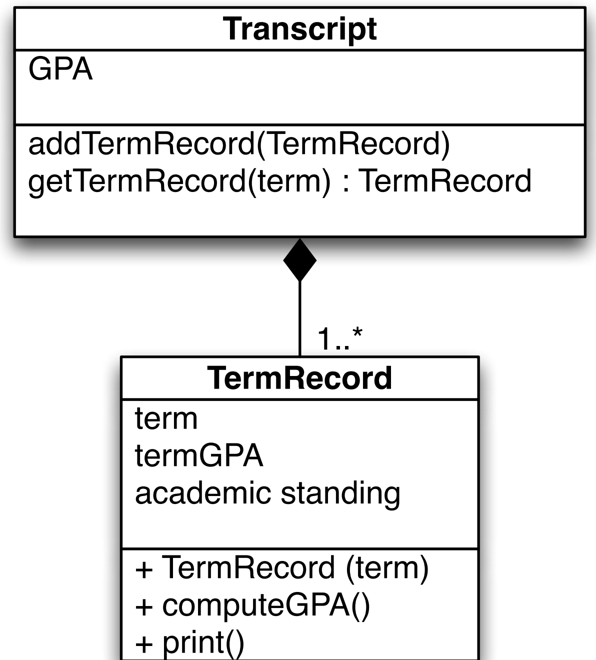
Client Code

```
Transcript *t;
TermRecord *tr;
...
tr = t->getTermRecord("Spring2014");
tr->computeGPA();
tr->print();
```

Composition and Data Encapsulation



vs.



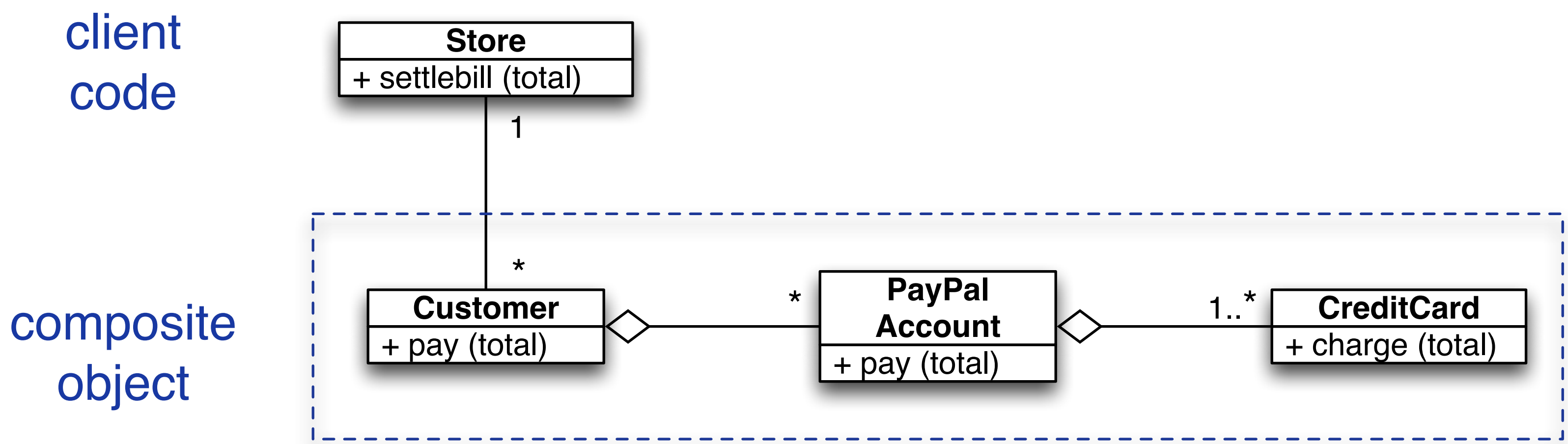
Client Code

Composite object offers methods for accessing and manipulating component information.

Client Code

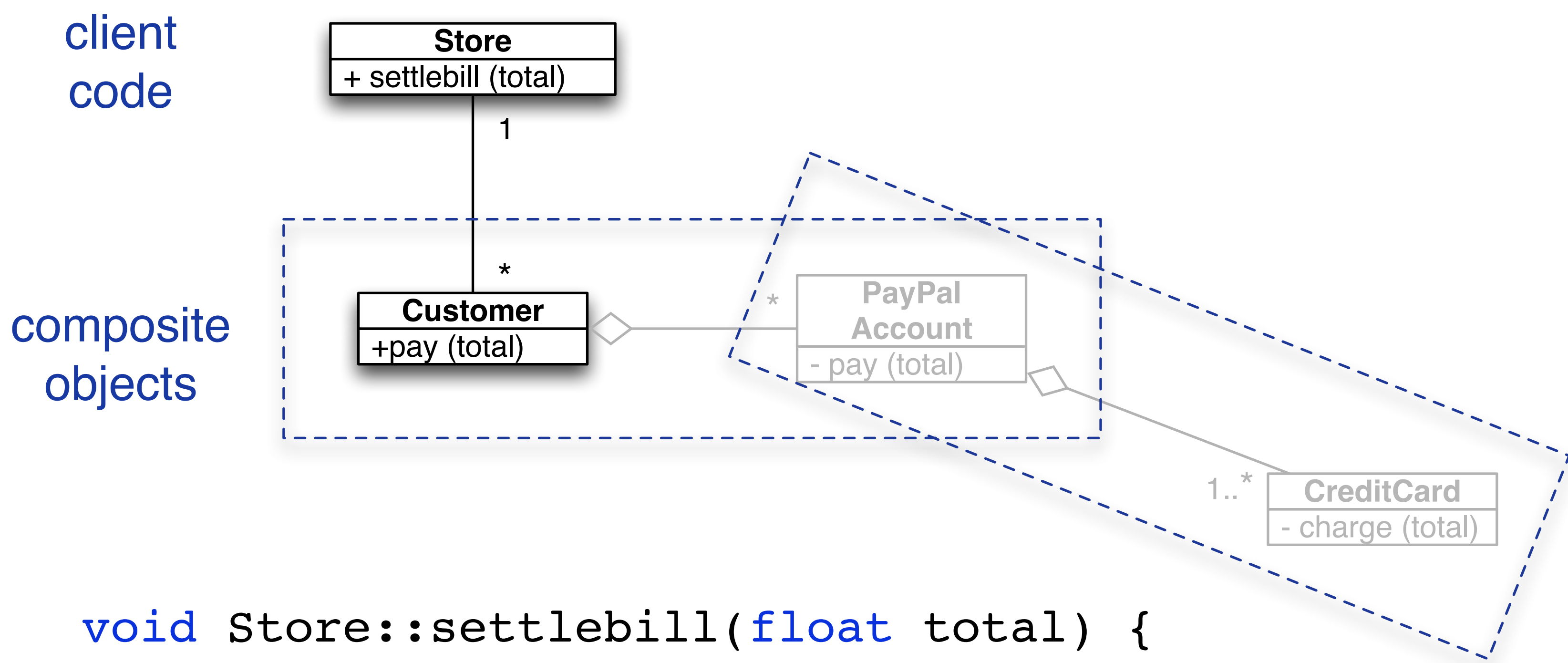
```
Transcript *t;
TermRecord *tr;
...
tr = t->getTermRecord("Spring2014");
tr->computeGPA();
tr->print();
```

Another Example



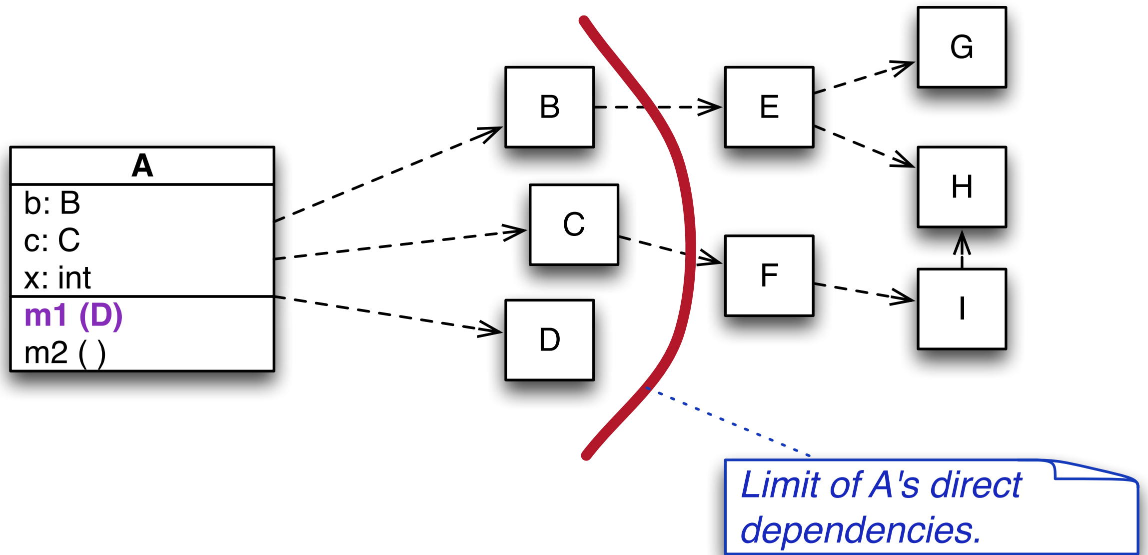

```
void Store::settlebill(float total) {
    . . .
    Customer->getPayPal()->getCreditCard()->charge(total);
    . . .
}
```

Component Encapsulation




```
void Store::settlebill(float total) {
    ...
    Customer->pay(total);
    ...
}
```

"Law" of Demeter



"Law" tests encapsulation: an object "talks only to its neighbours"

Method A::m1 can only call methods of

- A itself
- A's data members
- m1's parameters
- any object created by m1
- not methods of other classes (unless already allowed by above conditions)
  (e.g., not methods of an object simply because object is returned by a method call)

What is Wrong with this Design?