CS 247: Software Engineering Principles

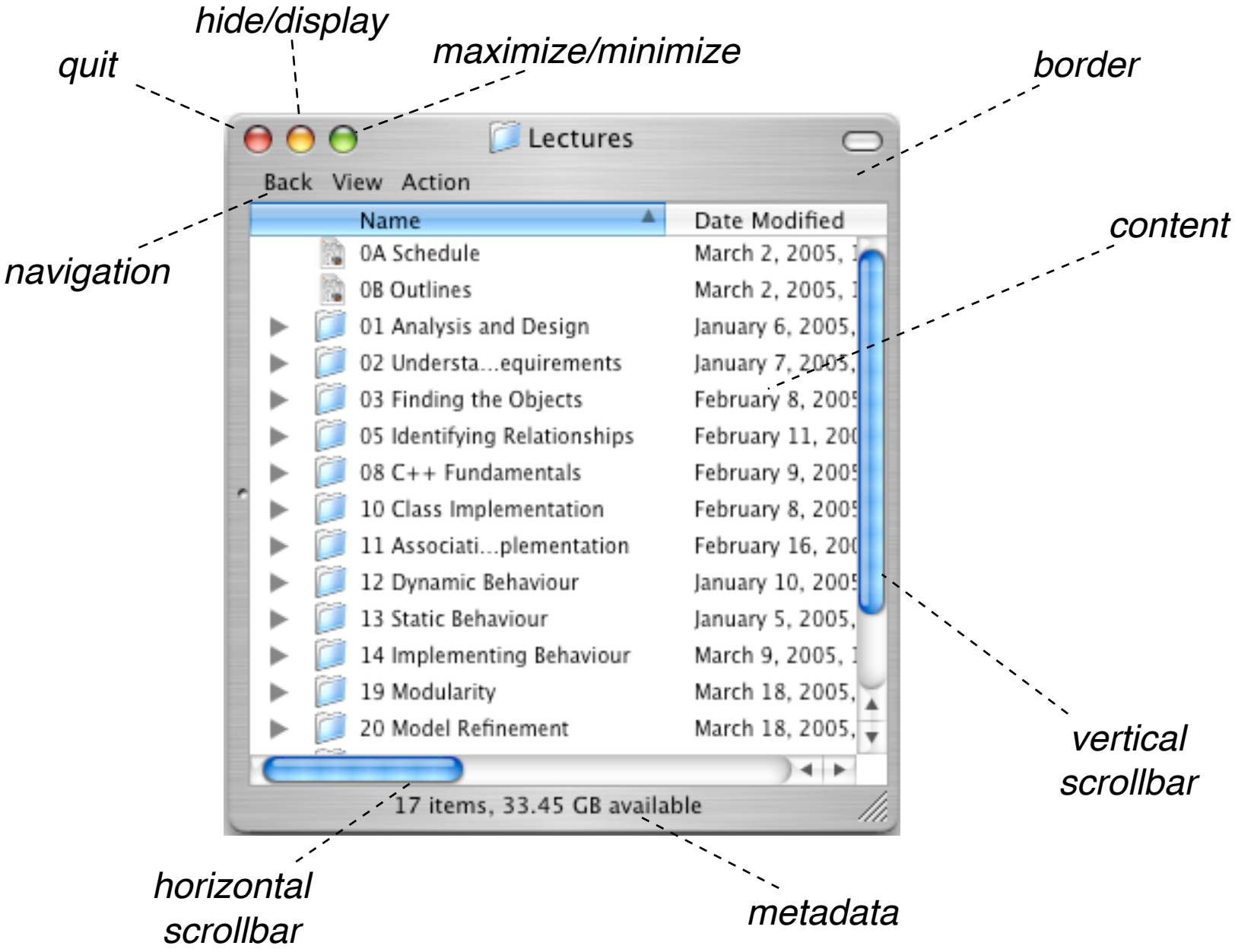
Design Patterns (Decorator, Factory)

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

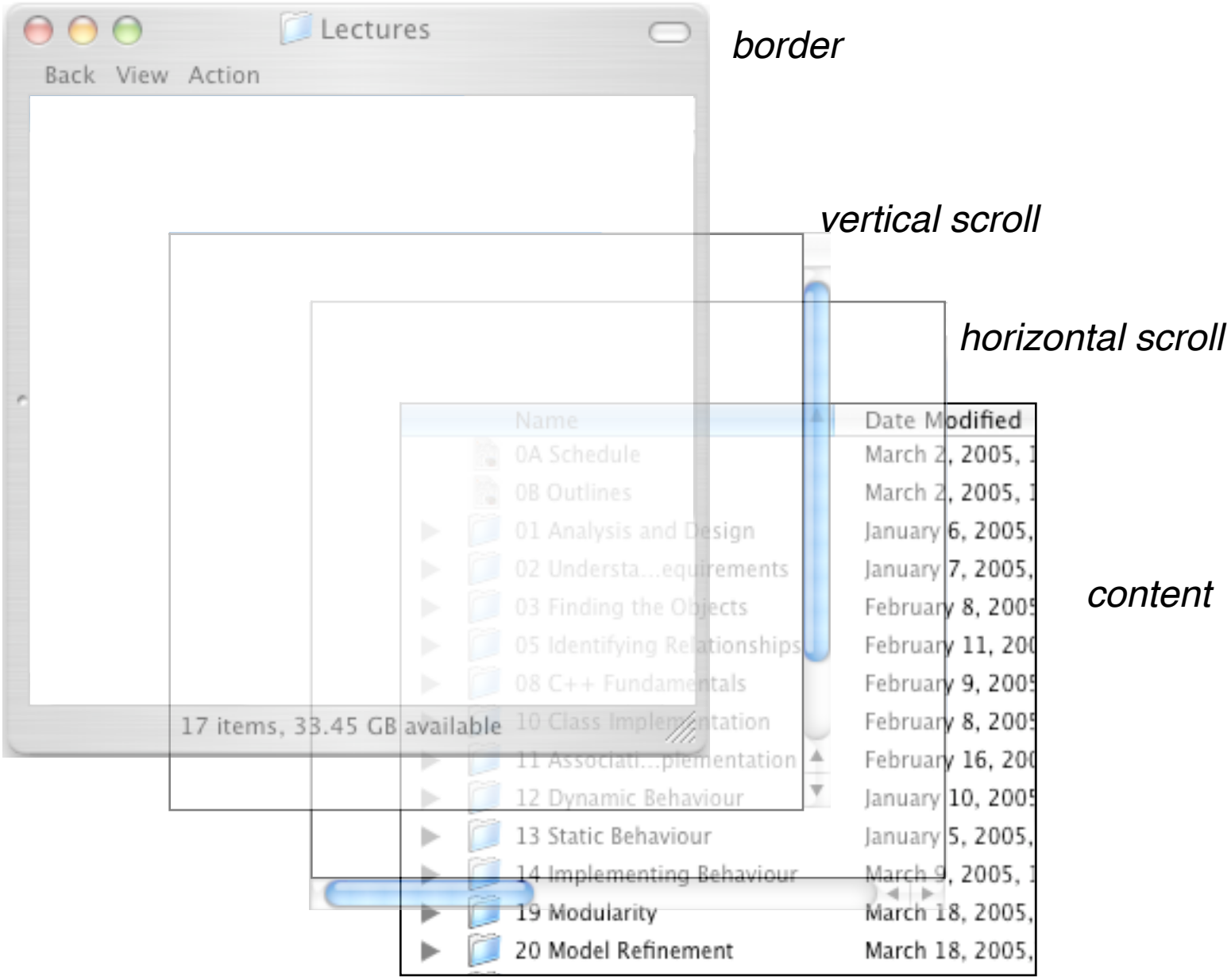
Ch 3 Decorator Pattern

Ch 4 Factory Method Pattern

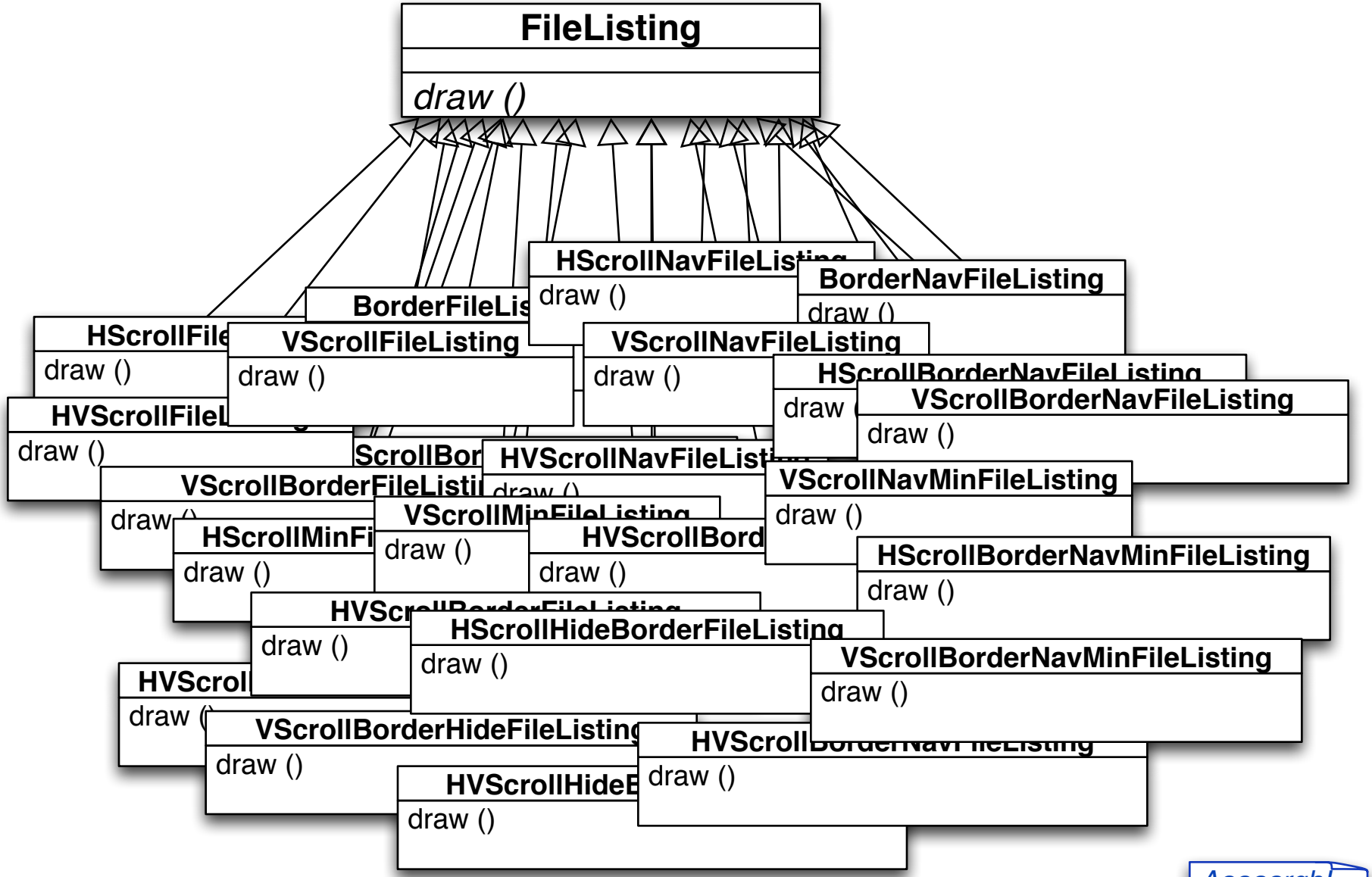
Problem: Window Appearance



Independent Features

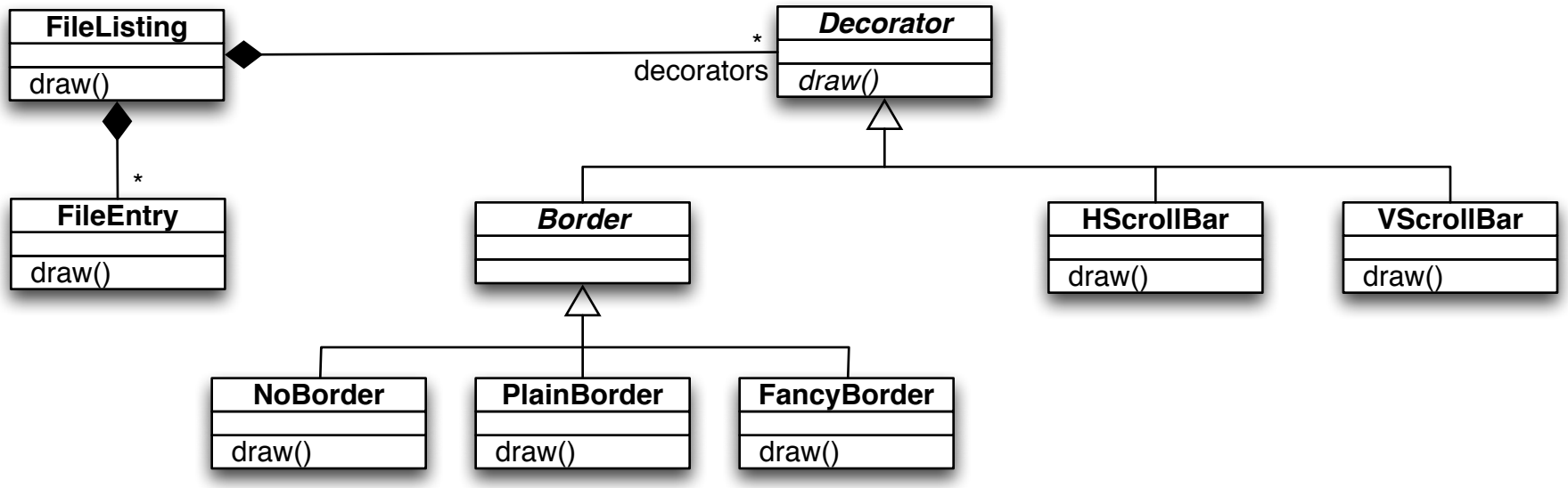


Solution 1: Inheritance



Solution 2: Composition and Interfaces

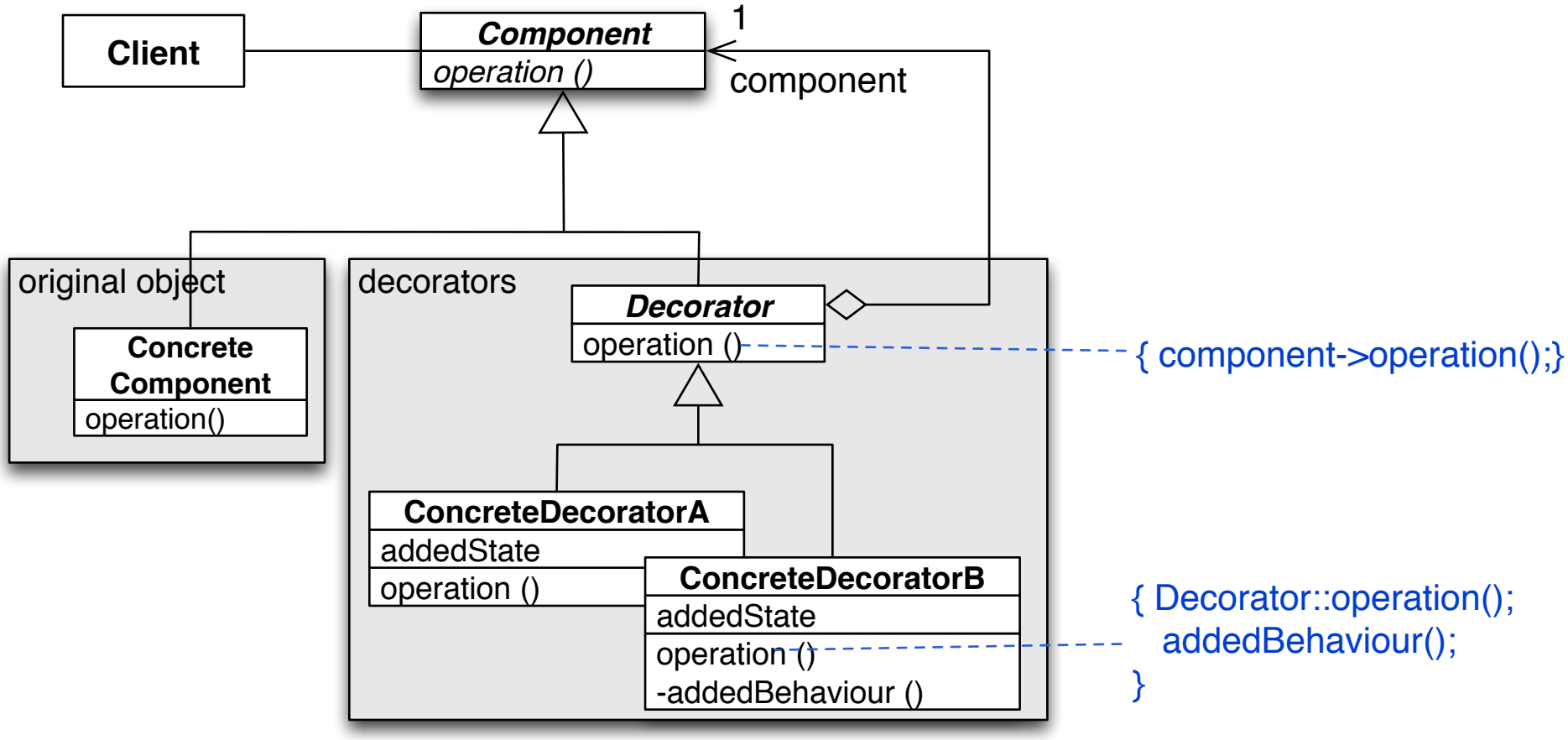
Composition and the programming to an interface design idiom lets us change window properties dynamically.



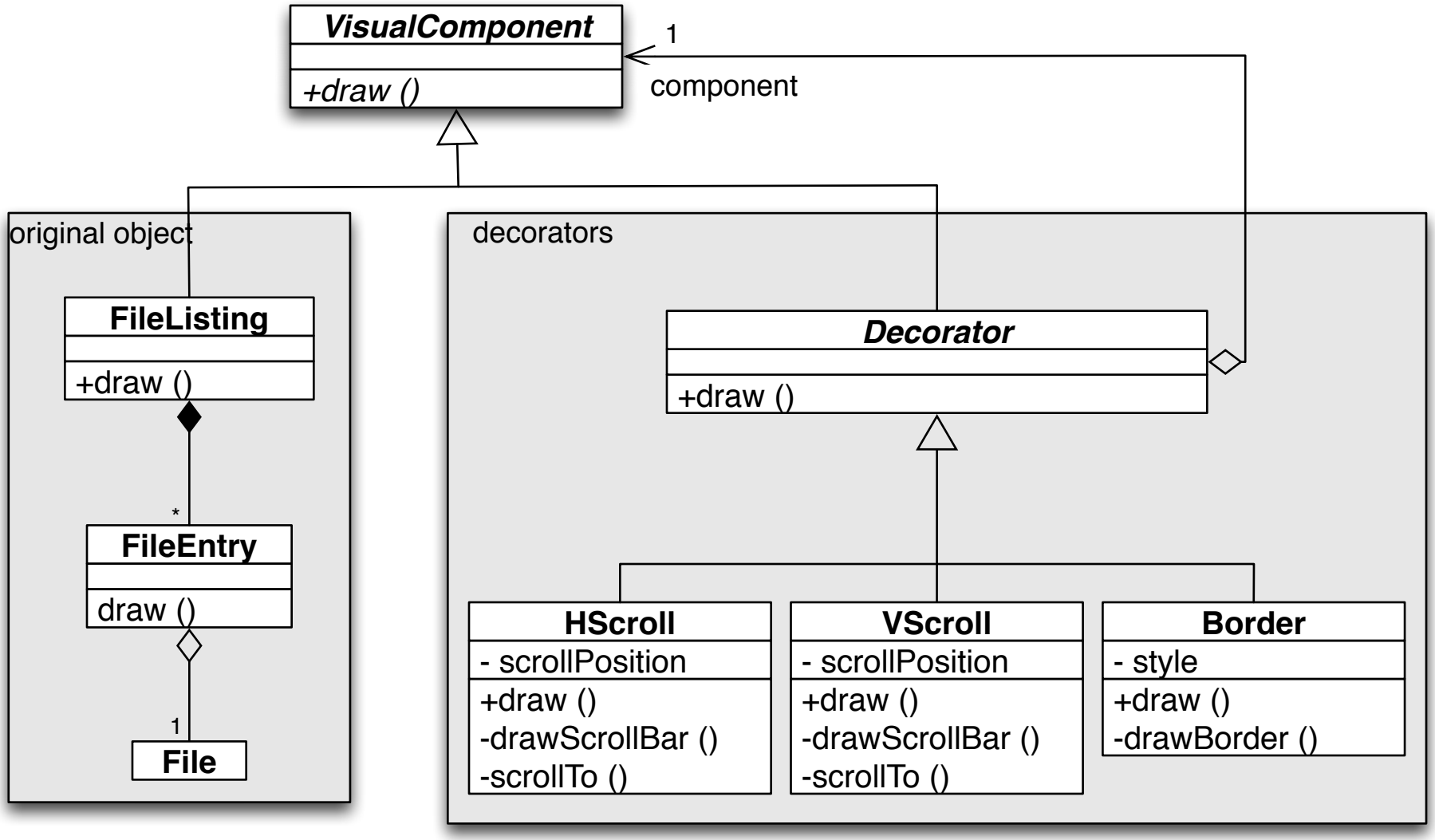
Decorator Pattern

Idea: Using composition, build a wrapper

A wrapper implements new functionality, and includes original object as a component.



Design Pattern Instantiated



Abstract Decorator

Shared decorator behaviour includes pointer to component, and original object's public operations.

```
class Decorator : public VisualComponent {
public:
    virtual void draw ();
protected:
    Decorator (VisualComponent* comp): component_ (comp) { }
private:
    VisualComponent* component_;
};
```

```
void Decorator::draw () {
   component_->draw ();
}
```

Concrete Decorator

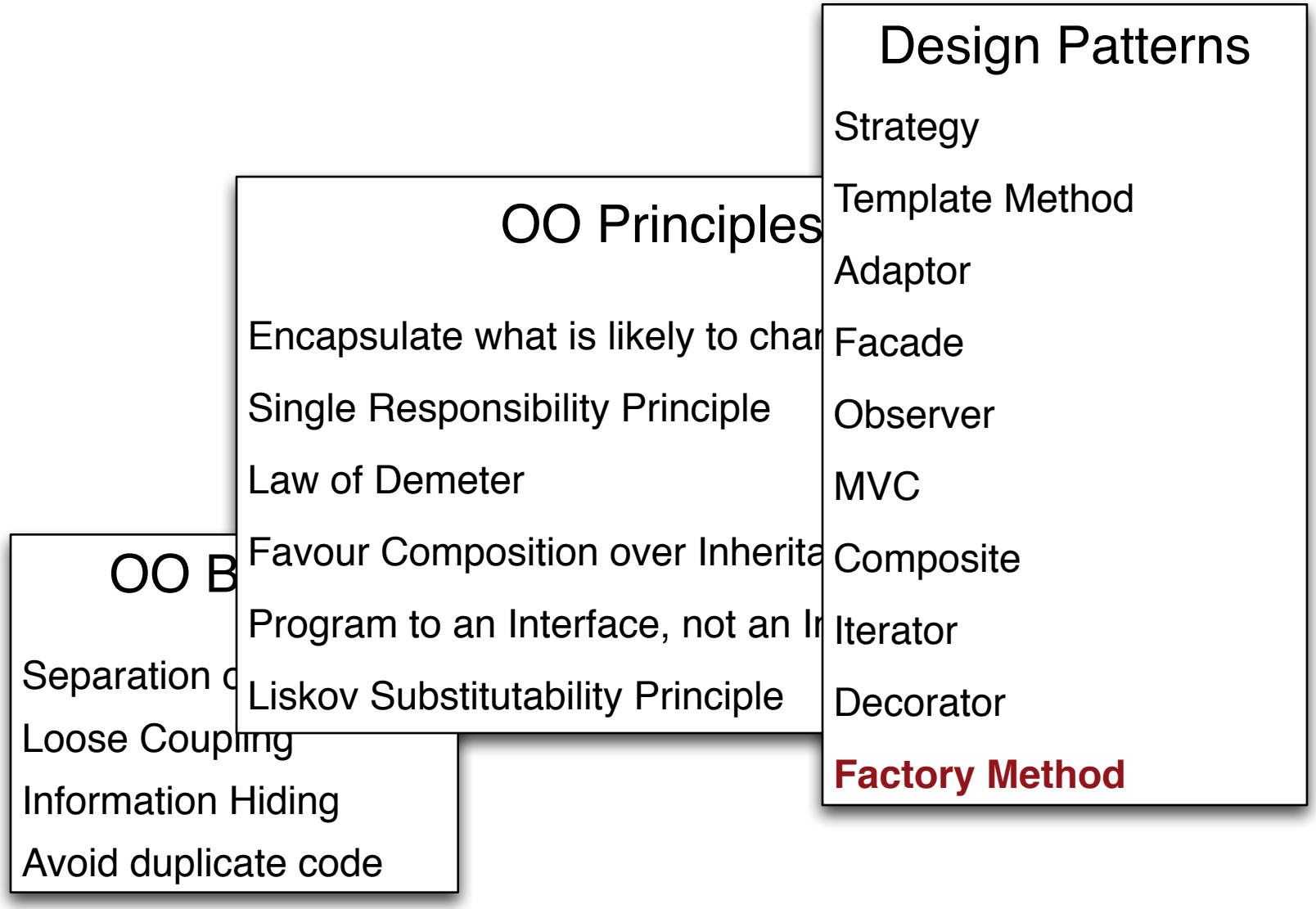
```
class Border : public Decorator {
    enum Style { DEFAULT, ... } style_;
public:
    Border (VisualComponent *v, Style s = DEFAULT):
        Decorator(v), style_ (s) {}
    void draw ();
};
```

```
void Border::draw () {
    Decorator::draw ();
    // code that draws border
}
```

Summary

Decorator Pattern: encapsulates "features" or additional responsibilities or functionality that can be added to a class at runtime

Design Patterns



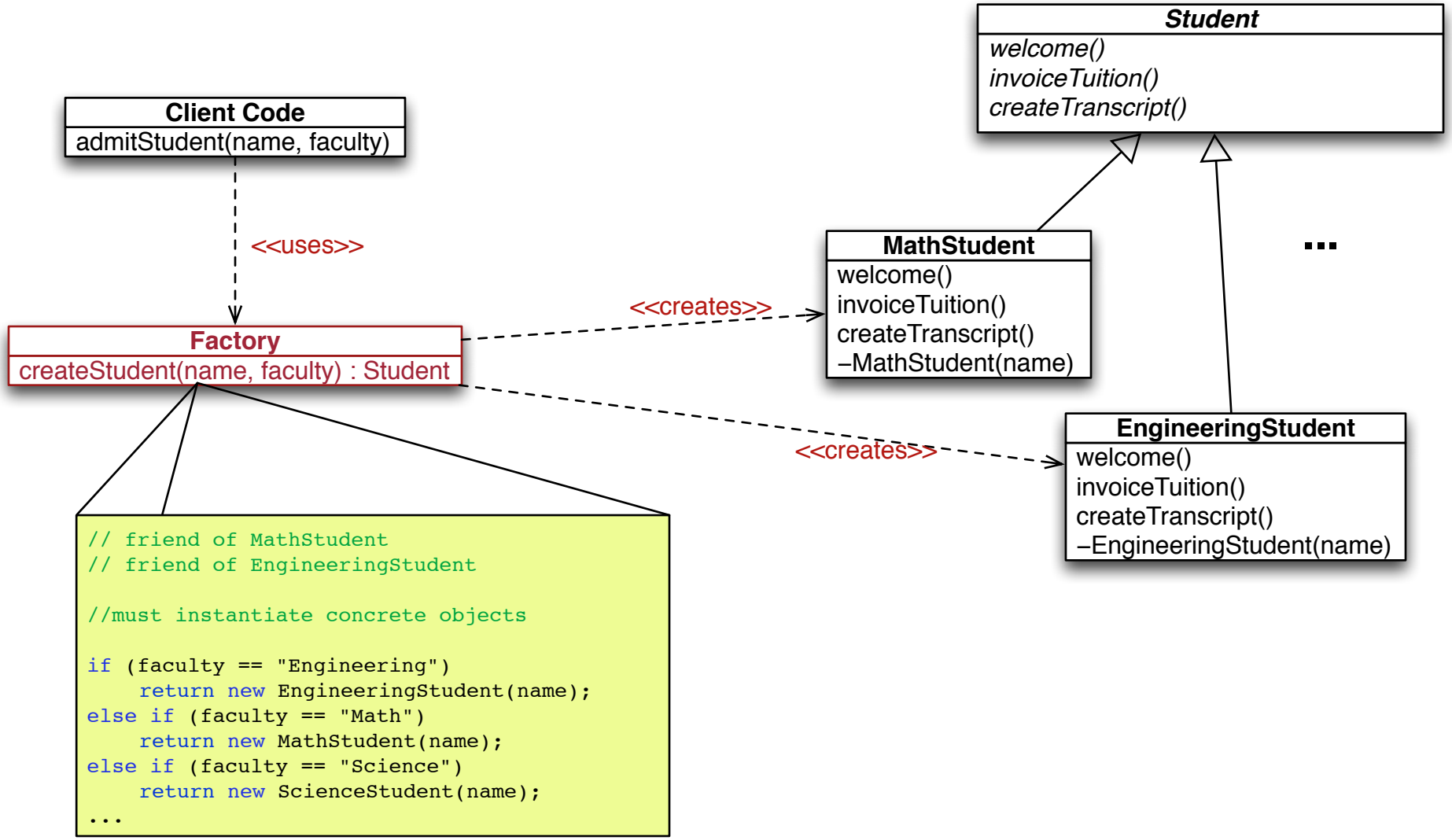
Instantiating Objects (of Concrete Type)

Program to an Interface, not Implementation is hard to follow when creating new objects -- cannot instantiate abstract objects.

```
void admitStudent (const string &name, const string &faculty)
{
        Student *s;

        // must instantiate concrete objects
        if (faculty == "Engineering")
           s = new EngineeringStudent(name);
        else if (faculty == "Math")
           s = new MathStudent(name);
        else if (faculty == "Science")
           s = new ScienceStudent(name);
...
   // Each student type has its own admission operations
   s->welcome();
   s->invoiceTuition();
   s->createTranscript();
}
```

Approach 1: Encapsulation

Encapsulate code that creates concrete objects in a Simple Factory. Not a design pattern.



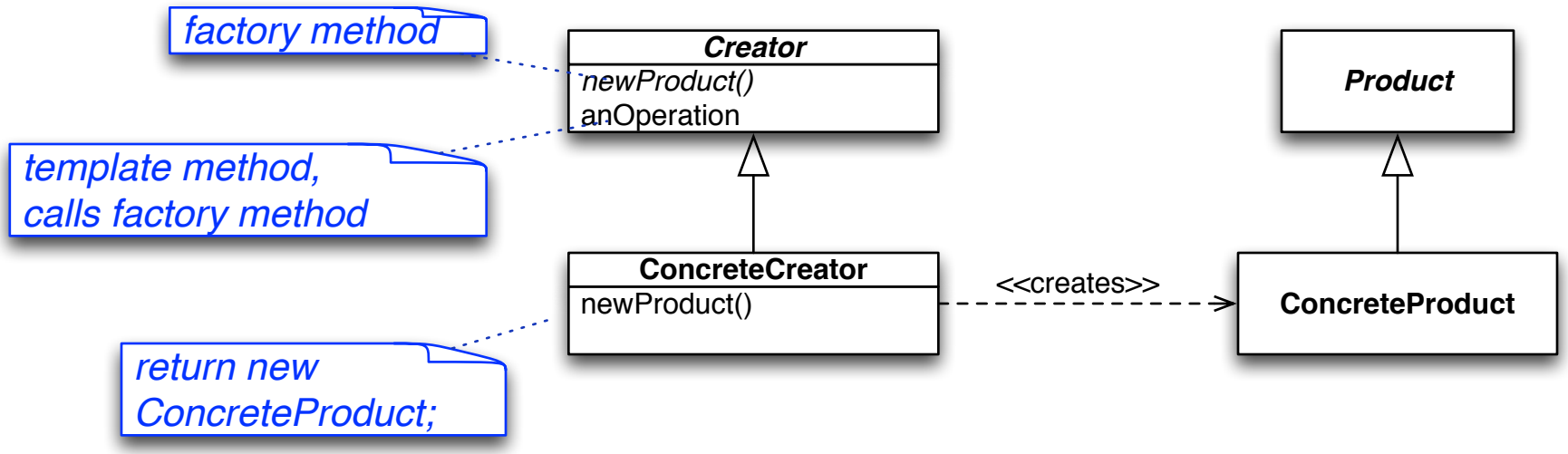
Approach 2: Factory Method Pattern

Problem: encapsulate the code that creates concrete objects

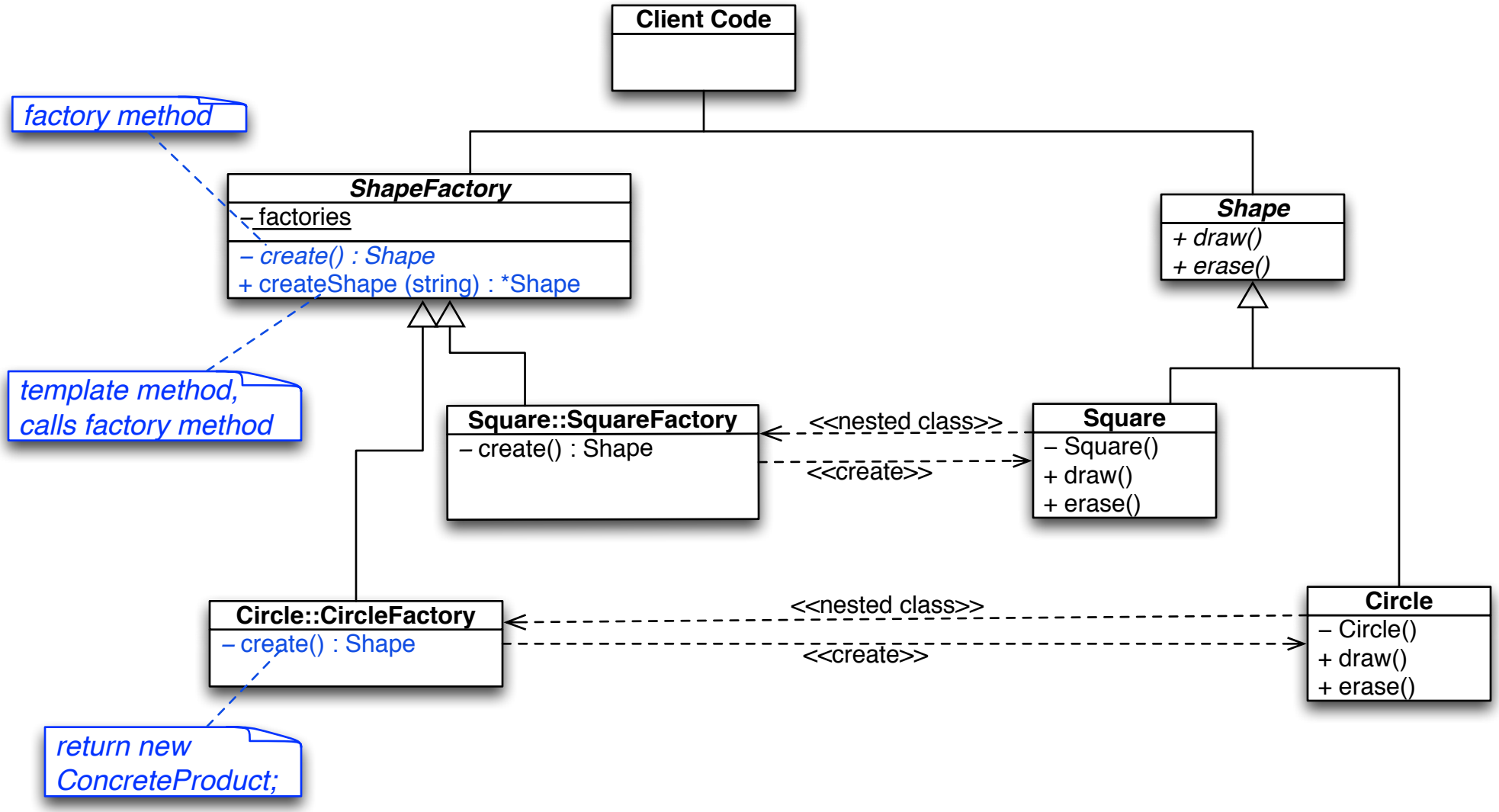
- Factories are polymorphic

Solution: use the Template Method

- Abstract class defines a method (template method)
- Factory method is a primitive operation of the template method
- Subclasses override factory method to construct specific concrete objects



Factory Method (Instantiated)



Polymorphic Factory

From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.

```
class ShapeFactory {
  virtual Shape* create() = 0; // Factory Method
  static std::map<std::string, ShapeFactory*> factories_;

public:
  virtual ~ShapeFactory() {}
  friend class ShapeFactoryInitializer;

  static Shape* createShape( const std::string& id ) {
    if ( factories_.find(id) != factories_.end() )
      return factories_[id]->create();
  }
};

std::map<std::string, ShapeFactory*> ShapeFactory::factories_;
```

Concrete Classes

From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.

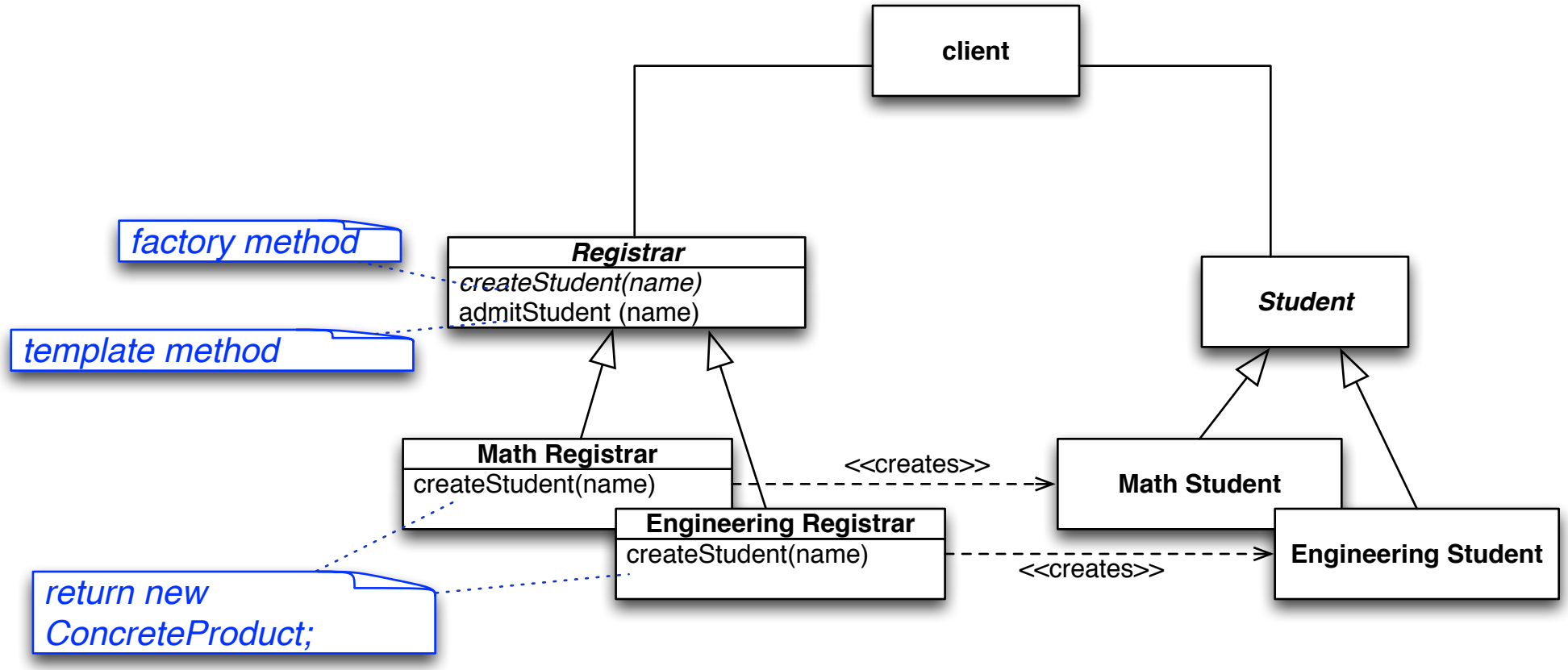
```
class Circle : public Shape {
  Circle() {} // Private constructor
  friend class ShapeFactoryInitializer;

  class Factory;
  friend class Factory;
  class Factory : public ShapeFactory {
  public:
    Shape* create() { return new Circle; }
    friend class ShapeFactoryInitializer;
  };

public:
  void draw() { cout << "Circle::draw" << endl; }
  void erase() { cout << "Circle::erase" << endl; }
  ~Circle() { cout << "Circle::~Circle" << endl; }
};
```

Factory Initialization

From "Thinking in C++, Volume 2", by Bruce Eckel & Chuck Allison.

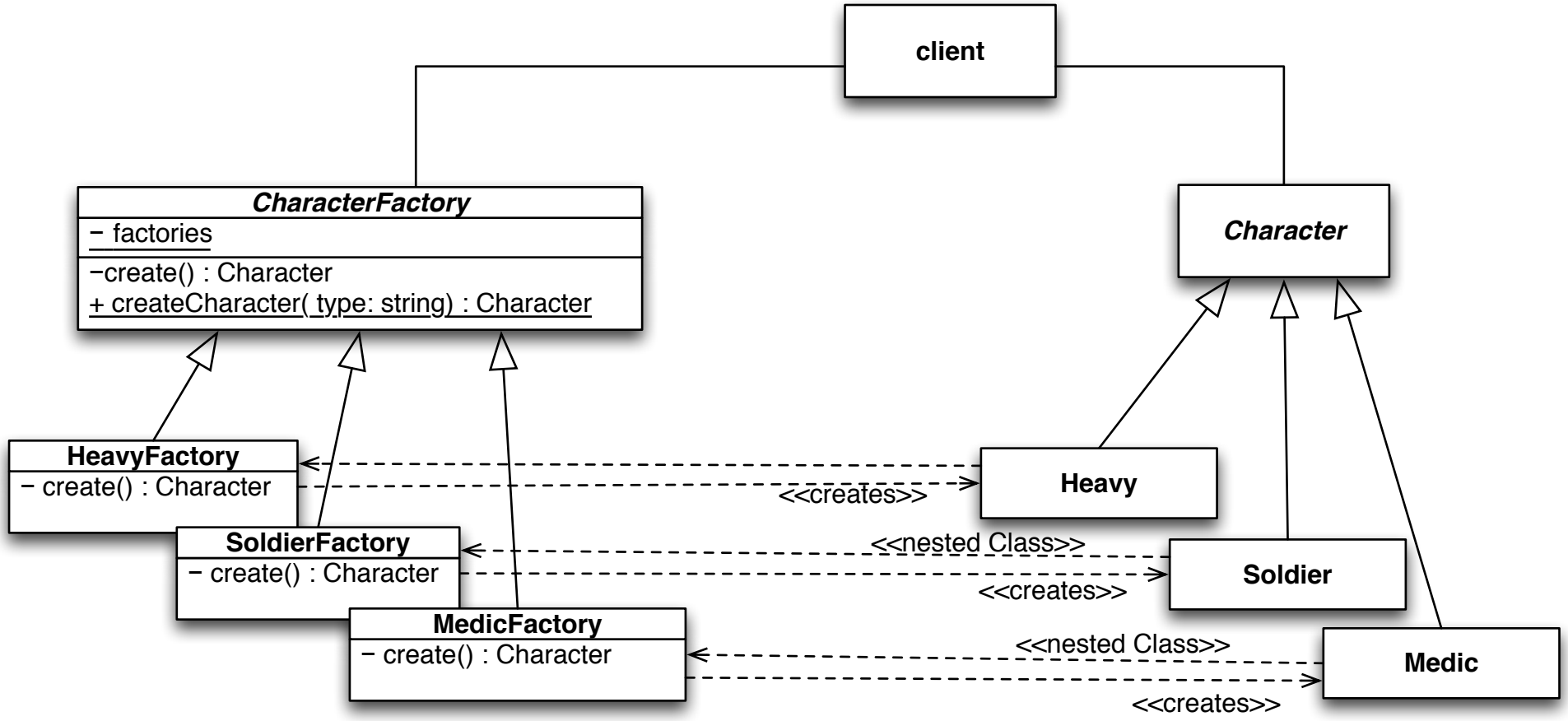
```
// Singleton to initialize the ShapeFactory:
class ShapeFactoryInitializer {
  static ShapeFactoryInitializer si;
  ShapeFactoryInitializer() {
    ShapeFactory::factories_["Circle"]= new Circle::Factory;
    ShapeFactory::factories_["Square"]= new Square::Factory;
  }

  // Destructor deletes all factories
  ~ShapeFactoryInitializer() {
    map<string, ShapeFactory*>::iterator it =
      ShapeFactory::factories_.begin();
    while(it != ShapeFactory::factories_.end())
      delete it++->second;
  }
};

// Static member definition:
ShapeFactoryInitializer ShapeFactoryInitializer::si;
```

Factory Method Pattern (Instantiated) 2



Factory Method Pattern (Instantiated) 3

From Xinhao Tian, SE_2014



Summary

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory	families of product objects
	Builder	how a composite objects gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class
Structural	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Facade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location
Behavioural	Chain of Responsibility	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed, traversed
	Mediator	how and which objects interact with each other
	Memento	what private information is stored outside an object, and when
	Observer	objects that depend on another object; how the dependent objects stay up to date
	State	states of an object
	Strategy	an algorithm
	Template Method	steps of an algorithm
	Visitor	operations that can be applied to objects(s) without changing their class(es)

Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1995 (p.30)