

Design patterns

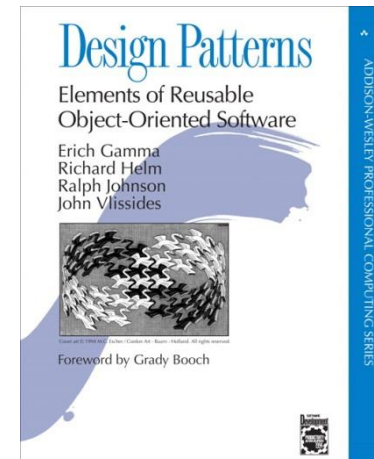
- the theory behind the practice -

Claudio Scordino,
Software Engineer, PhD
Version 1.23

Disclaimer

Claudia Scardine

- Most content of this talk comes from
 - Giuseppe Lipari's course
(<http://retis.sssup.it/~lipari/courses/oosd2010-2/>)
 - The book "*Design Patterns*":



Theory...

Object-oriented design

Claudia Scardine

- Object-oriented languages ease the decomposition of the problem into a set of objects
- During the design, it is important to:
 - identify which are the objects
 - identify which services they provide each other
- Objects can be thought as **service providers**
- Objects can be always decomposed into smaller objects. We should stop when objects are "*small enough*":
 - they can be easily implemented
 - they are self-consistent and self-contained

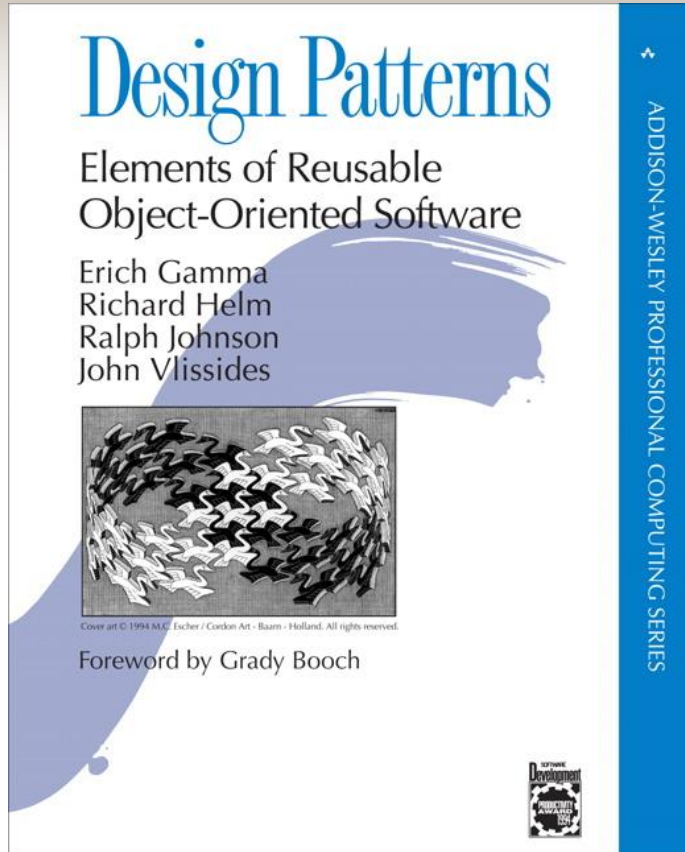
Design patterns

Design pattern: general reusable solution to a commonly occurring problem in software design

- It is a description or template for how to solve a problem that can be used in many different situations
 - Patterns \equiv documented experience
- Design patterns are best exploited with object-oriented languages (e.g., C++) even if the same behavior can be implemented in procedural languages (like C)

Reference book

Claudia Scardine



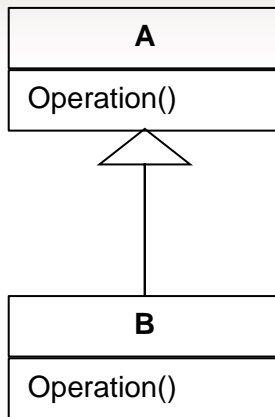
- Reference book written by the "Gang of Four" (GoF) in 1994
- Presents 23 patterns divided in 3 categories
- It also presents a common design jargon (factory, delegation, composite, etc.)

Reference book (2)

Claudia Seardine

- Inspired by a book on architecture design: *"each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"*.
- From architecture to software design: one of the few examples in which software development has been inspired by other areas of engineering

Book's notation

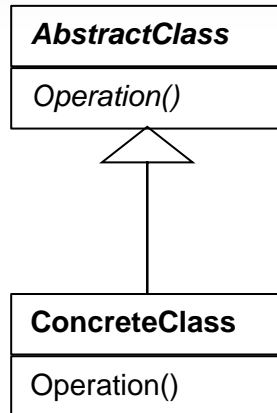


```
class A {
    //...
};

class B: public A {
    //...
};
```


Book's notation (2)

Claudia Seardine

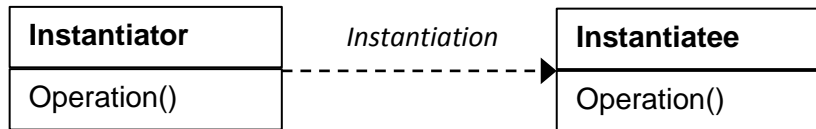


```
class AbstractClass {
public:
    virtual void Operation() = 0;
    //...
};

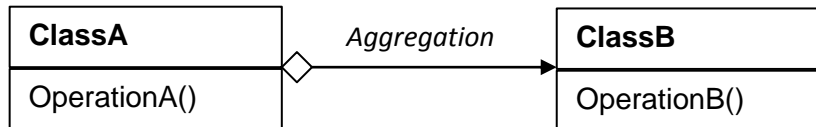
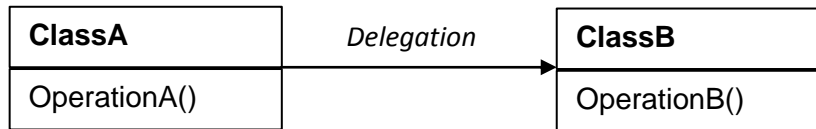
class ConcreteClass: public A {
public:
    virtual void Operation() {...}
    //...
};
```

Book's notation (3)

Claudia Scardine



```
void Instantiator::function()
{
    //...
    inst = new Instantiateee;
```



Basic ideas

Claudia Seardine

- Some basic principles allow us to create systems that are easier to maintain and to extend
- Key ideas:
 - **Composition**: used for extending functionality
 - **Inheritance**: used to make interfaces uniform

1st GoF's principle

Claudia Scardine

*Program to an interface (i.e., an abstract class),
not to an implementation (i.e., a concrete class)*

Benefits:

- Clients remain unaware of the specific types of objects they use (as long as the objects adhere to the interface that clients expect)
- This principle reduces implementation dependencies between subsystems

2nd GoF's principle

Claudia Scardine

Favor object composition over class inheritance

Drawbacks of inheritance:

- Statically defined at compile-time
- It breaks encapsulation: subclasses exposed to parent's details
- Dependencies between parent and subclasses, which limits flexibility and reusability

The SOLID principles

Claudia Seardine

SOLID: Mnemonic acronym for 5 basic principles for good object-oriented design:

- S(RP): Single responsibility principle
- O(CP): Open/closed principle
- L(SP): Liskov substitution principle
- I(SP): Interface segregation principle
- D(IP): Dependency inversion principle

S: Single Responsibility Principle

Claudia Seardine

*A class should have only one responsibility
(AKA "reason to change")*

- When we need to make a change in a class having more responsibilities, the change might affect the other functionality of the class.
- Rule: keep each class focused on a single concern!

S: Single Responsibility Principle

Claudia Scardine

Too many responsibilities on a single thing can cause problems.



S: Single Responsibility (example)

Claudia Seardine

```
class Camera
{
public:
    void set_user(const std::string& user);
    void set_passwd(const std::string& user);
    void set_url(const std::string& user);
    image take_snapshot();
    void setBrightness(int value);
private:
    std::string user;
    std::string passwd;
    std::string url;
};
```



S: Single Responsibility (example 2)

Claudia Seardine



```
class Session
{
public:
    Session(const std::string& user,
            const std::string& passwd,
            const std::string& url);

private:
    std::string user_;
    std::string passwd_;
    std::string url_;

};
```

```
class Camera
{
public:
    Camera(Session);
    image take_snapshot();
    void setBrightness(int value);

private:
    Session session_;

};
```

O: Open/Closed Principle

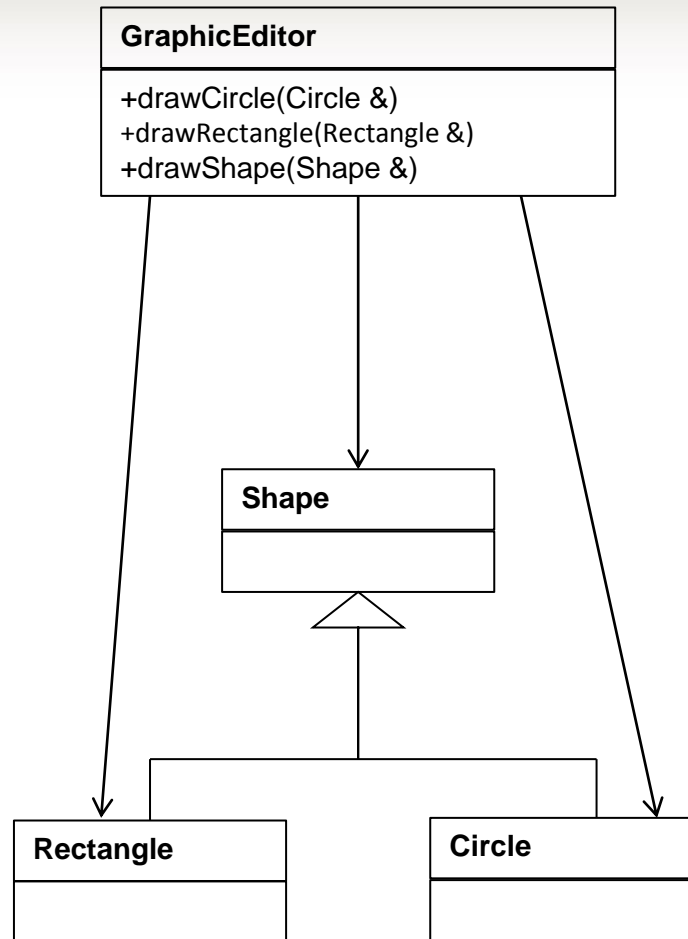
Claudia Seardine

*A class should be open for extension
but closed for modification*

- Every software is subject to change
 - A good design makes changes less trouble
 - Keyword: *prepare for change*
- Ideally, no need of changing existing code to add new features or extensions (i.e., only for bug-fixing and maintainance)
- New features should be added by either:
 - adding new subclasses and overriding methods, or
 - reusing existing code through delegation

A bad design

Claudia Scardine



A bad design (code)

Claudia Scardine

```
class GraphicEditor {
public:
    void drawShape(Shape &s) {
        if (s.type==1) drawRectangle(s);
        else if (s.type==2) drawCircle(s);
    }
    void drawCircle(Circle &r) {....}
    void drawRectangle(Rectangle &r) {....}
};

class Shape {
public:
    int type;
};

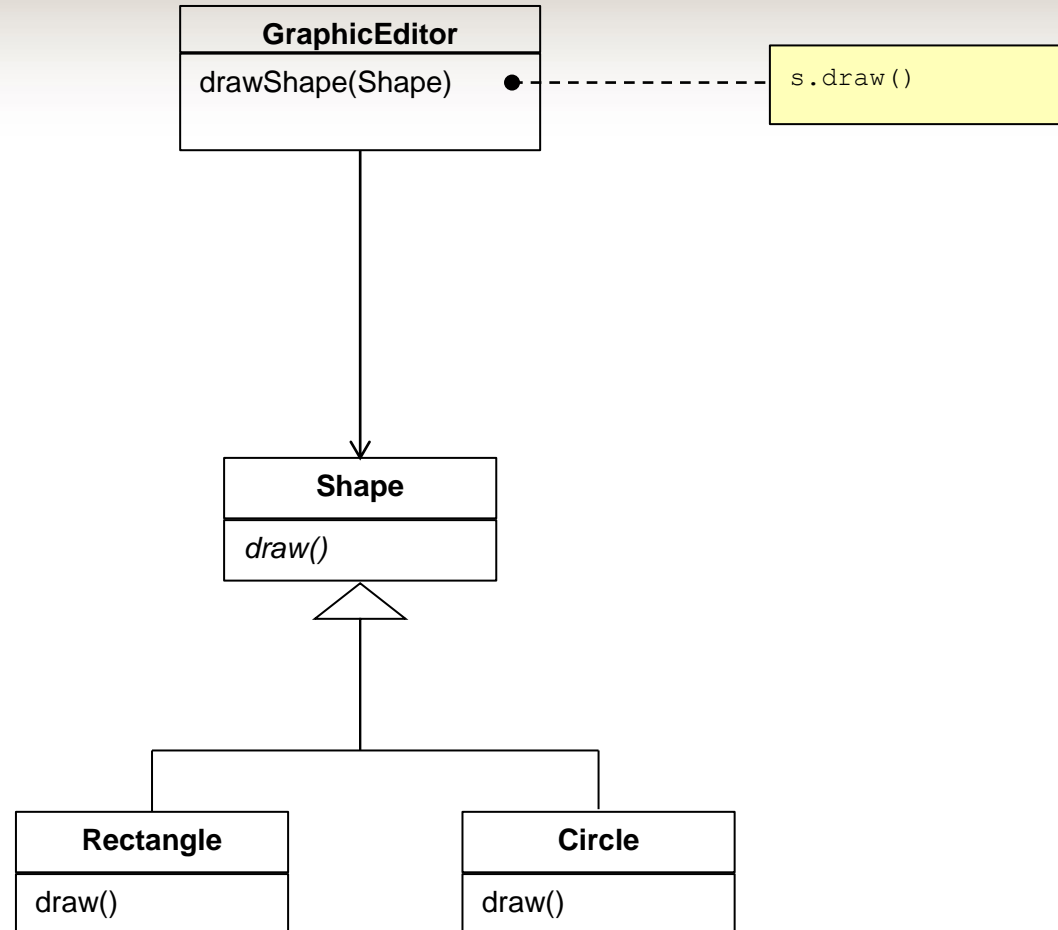
class Rectangle : public Shape {
    Rectangle() { type=1; }
};

class Circle : public Shape {
    Circle() { type=2; }
};
```



A good design

Claudia Scardine



L: Liskov Substitution Principle

Claudia Seardine

Functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it

- Principle stated by Barbara Liskov
- The importance of this principle becomes obvious when you consider the consequences of violating it.
- If a function which does not conform to the LSP, then it must know all the possible derivatives of the base class.

L: Liskov Substitution Principle (2)

Claudia Seardine

Example of violation:

```
void DrawShape(const Shape& s)
{
    Square *q;
    Circle *c;
    if (q == dynamic_cast<Square *>(s))
        DrawSquare(q);
    else if (c == dynamic_cast<Circle *>(s))
        DrawCircle(c);
}
```

- The DrawShape function is badly formed: it must know about every possible derivative of the Shape class, and it must be changed whenever new derivatives of Shape are created.

L: Liskov Substitution Principle (3)

Claudia Seardine

Example of a more subtle violation...

```
class Rectangle
{
public:
    virtual void SetWidth(const double w) {width_=w;}
    virtual void SetHeight(const double h) {height_=w;}
    virtual double GetHeight() const {return height_;}
    virtual double GetWidth() const {return width_;}
private:
    double width_;
    double height_;
};
```

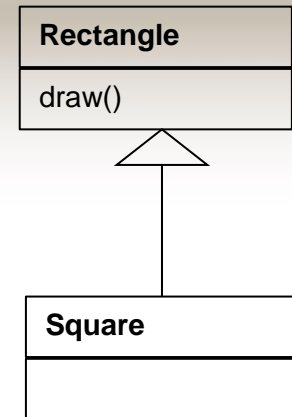
- We now want to introduce a Square
- A square is a particular case of a rectangle, so we decide to derive class Square from class Rectangle

L: Liskov Substitution Principle (4)

Claudia Seardine

```
class Square : public Rectangle
{
public:
    virtual void SetWidth(const double w) {
        Rectangle::SetWidth(w);
        Rectangle::SetHeight(w);
    }
    virtual void SetHeight(const double h) {
        Rectangle::SetWidth(w);
        Rectangle::SetHeight(w);
    }
}
```

```
void function (Rectangle& r)
{
    r.SetWidth(5);
    r.SetHeight(4);
    assert(r.GetWidth() * r.GetHeight()) == 20);
}
```



**This function does not work
with Squares!**

I: Interface Segregation Principle

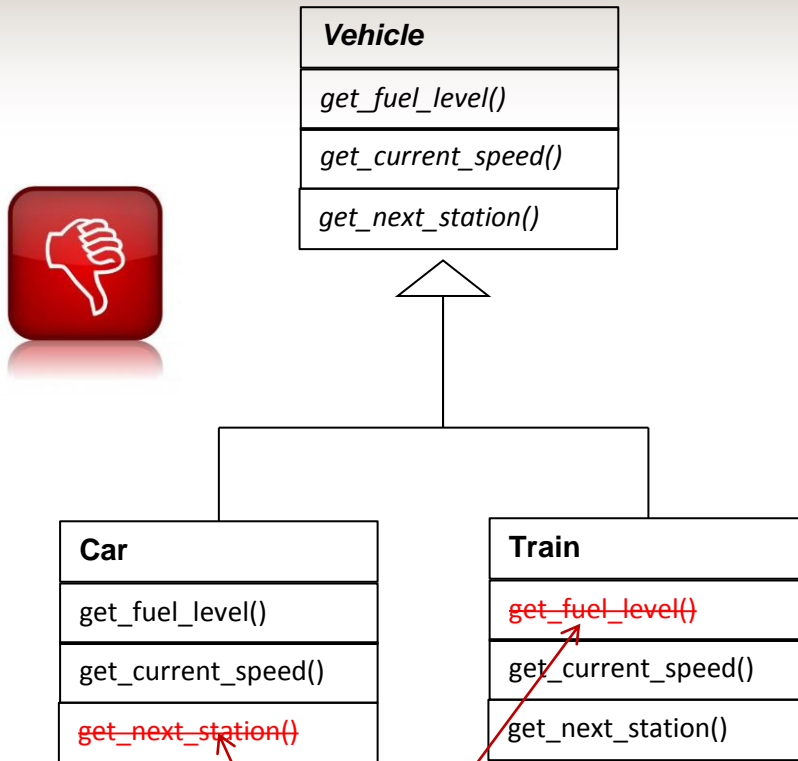
Claudia Seardine

Clients should not be forced to depend upon interfaces that they don't use

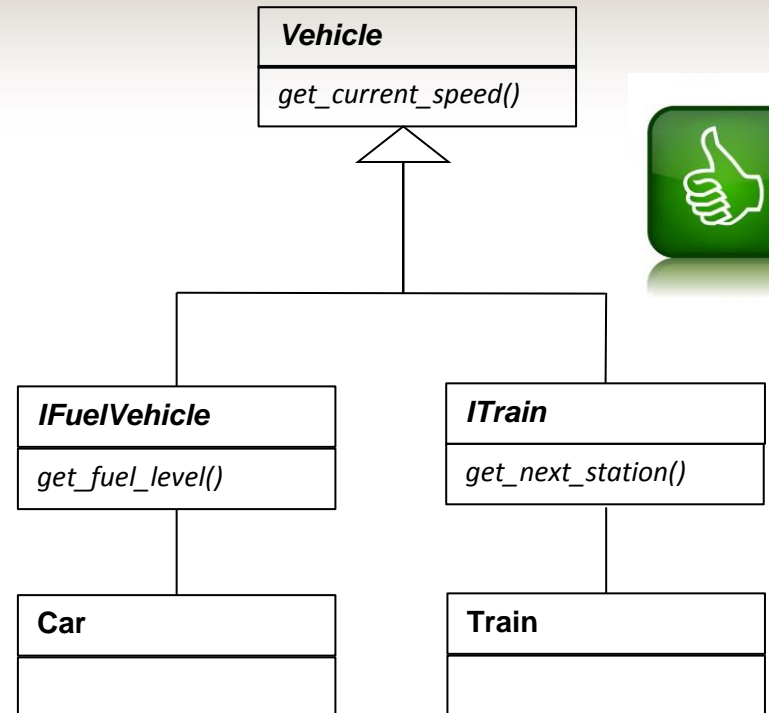
- When we write our interfaces we should take care to add only methods that should be there. Otherwise, subclasses will have to implement those methods as well.
- Use small interfaces
- Avoid polluted or fat interfaces

I: Interface Segregation (example)

Claudia Scardine



Not implemented.
(throw exceptions ?)



D: Dependency Inversion Principle

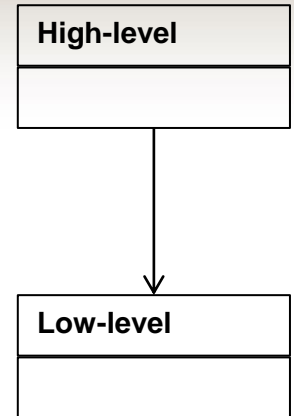
Claudia Seardine

1. *High-level modules should not depend on low-level modules.
Both should depend on abstractions.*
2. *Abstractions should not depend upon details.
Details should depend upon abstractions.*

D: Dependency Inversion Principle (2)

Claudia Seardine

- In an application we usually have:
 - low-level classes which implement basic operations
 - high-level classes which encapsulate complex logic and rely on the low level classes.



- Natural way of development: write low-level classes and then write complex high-level classes
- This is not a flexible design: what happens if we need to replace a low-level class?

D: Dependency Inversion Principle (3)

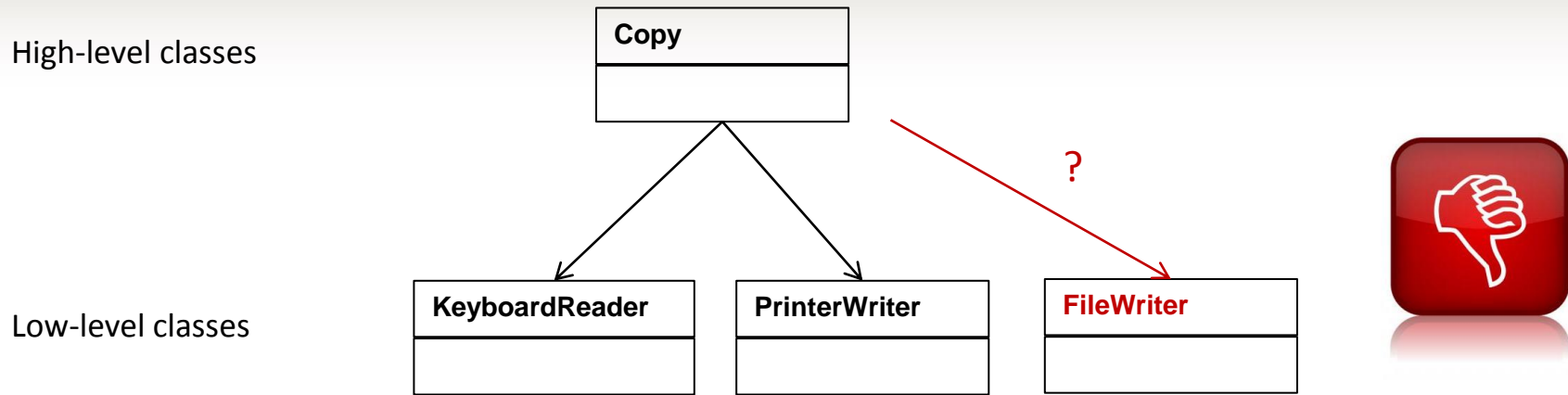
Claudia Seardine

Example:

- A copy module which reads characters from keyboard and write them to the printer device.
- The high-level class containing the logic is the `Copy` class.
- The low-level classes are `KeyboardReader` and `PrinterWriter`.
- In a bad design the high level class uses directly the low level classes. If we want to change the design to direct the output to a new `FileWriter` class we have to change the `Copy` class.
- Since the high level class contains the complex logic, it should not depend on the low-level classes
- An abstraction layer should be created to decouple the two levels

D: Dependency Inversion (example)

Claudia Scardine



Dependency inversion: a high-level class (containing the complex logics) depends on the low-level classes

D: Dependency Inversion Principle (4)

Claudia Seardine

- According to this principle, the way of designing a class structure is to start from high-level modules to the low-level modules:

High-level Classes → Abstraction Layer → Low-level Classes

This can be achieved by:

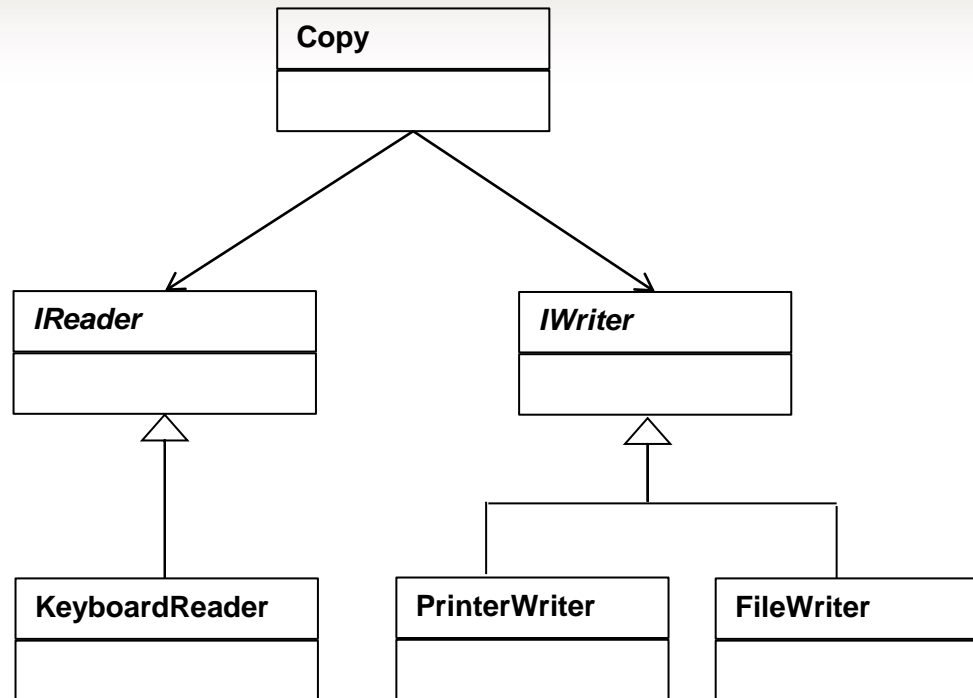
- writing abstract interfaces to low level classes (i.e., abstract layer)
- making sure that high-level classes use only references to the abstract interfaces
- using some creational pattern to make the connection

D: Dependency Inversion (example)

Claudia Scardine

High-level classes

Low-level classes



... and practice...

Classification of Design Patterns

Claudia Seardine

Design patterns are extensive applications of these principles

GoF divides design patterns in:

- **Creational** patterns: concern object creation
- **Structural** patterns: deal with the composition of classes/objects
- **Behavioral** patterns: characterize how classes/objects interact and distribute responsibilities
 - They describe patterns of message communications and are concerned with *algorithms* rather than with structures.

Creational patterns

Creational patterns

Claudia Seardine

- **Abstract Factory**: provide an abstract interface for creating families of related or dependent objects without specifying their concrete classes
- **Builder**: Separate the construction of a complex object from its representation so that the same construction process can create different representations; useful whenever the creation of an object is complex and requires many different steps.
- **Factory method** (AKA Virtual Constructor): define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
- **Prototype**: specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Singleton**: ensure a class only has one instance, and provide a global point of access to it.

Creational patterns: Singleton

Singleton: intent

Claudia Scardine

Ensure that a class only has one instance, and provide a global point of access to it

- For some classes it is important to have exactly one instance
 - E.g., there should be only one window manager
- Of course, this can be achieved with a global variable
- However, for complex systems we could run in some problems (e.g., object created many times by mistake, namespace pollution)
- A better solution is to make the class itself responsible for creating and maintaining the unique instance

Singleton: include file

Claudia Seardine

Include file:

```
class SysParams {  
public:  
    static SysParams &getInstance();  
    // other non-static members  
private:  
    static SysParams *inst_;  
    // other non-static members  
    SysParams();  
    SysParams(const SysParams &);  
};
```

Pointer to the only instance

Constructor made private

Copy constructor hidden and not implemented

Singleton: source file

Claudia Seardine

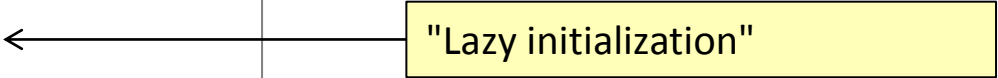
Source file:

```
SysParams *SysParams::inst_ = 0;

SysParams& SysParams::getInstance()
{
    if (inst_ == 0)
        inst_ = new SysParams();
    return *inst_;
}

SysParams::SysParams() { ... }
```

"Lazy initialization"



Singleton: benefits

Claudia Scardine

- Controlled access to the sole instance
- Deterministic initialization order of objects
- Reduced namespace (no pollution of namespace)
- Can be extended to control the number of instances created
- Typical example: Logger


Singleton: concurrency

Claudia Scordino

- If several threads can use the singleton, we must protect the initialization through a mutex semaphore
- To reduce overhead we use the double checked locking pattern:

```
SysParams& SysParams::getInstance()  
{  
    if (inst_ == 0) {  
        lock_mutex();  
        if (inst == 0) inst = new SysParams();  
        unlock_mutex();  
    }  
    return *inst_;  
}
```

Double checked locking pattern



- Warning: on some architectures you need a memory barrier due to instructions re-ordering (http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf)

Meyers Singleton

Claudia Scardine

- Implementation by Scott Meyers
- Thread-safe in C++11 (§6.7 [stmt.dcl] p4)

```
static SysParams& SysParams::getInstance()  
{  
    static SysParams s;  
    return s;  
}
```



Creational patterns: Abstract Factory

Abstract Factory: motivation

Claudia Scardine

- A program must be able to choose one of several families of classes
- Example:
 - A program's GUI should run on several platforms and each platform comes with its own set of GUI classes (WinButton, WinScrollBar, WinWindow, MotifButton, MotifScrollBar, MotifWindow, pmButton, pmScrollBar, pmWindow, etc.)

Naive approach

Claudia Seardine

- We keep a global variable (or object) that represents the current window manager and the “look-and-feel” for all the objects
- Every time we create an object, we execute a switch/case on the global variable to see which object we must create:

```
enum {WIN, MOTIF, PM, ...} lf;  
  
...  
// need to create a button  
switch(lf) {  
  case WIN: button = new WinButton(...);  
    break;  
  case MOTIF: button = new MotifButton(...);  
    break;  
  case PM: button = new PmButton(...);  
    break;  
}
```


Problems with the naive approach

Claudia Seardine

- What happens if we need to add a new look-and-feel?
 - We must change a lot of code (for every creation, we must add a new case)
- How much code must we link?
 - Assuming that each look and feel is part of a different library, all libraries must be linked together
 - Large amount of code
- This solution is not compliant with the open/closed principle
 - Every time we add a new look and feel, we must change the code of existing functions/classes
- This solution does not scale

Objectives

Claudia Scardine

- Uniform treatment of every button, window, etc.
 - Once you define the interface, you can easily use inheritance
- Uniform object creation
- Easy to switch between families
- Easy to add a family

Solution: Abstract Factory

Claudia Scardine

Provide an abstract interface for creating families of related or dependent objects without specifying their concrete classes

Solution: Abstract Factory (2)

Claudia Seardine

- Define a **factory** (i.e., a class whose sole responsibility is to create objects):

```
class WidgetFactory {  
    Button* makeButton(args) = 0;  
    Window* makeWindow(args) = 0;  
    // other widgets...  
};
```

- Define a concrete factory for each of the families:

```
class WinWidgetFactory : public WidgetFactory {  
    Button* makeButton(args) {  
        return new WinButton(args);  
    }  
    Window* makeWindow(args) {  
        return new WinWindow(args);  
    }  
};
```

Solution: Abstract Factory (3)

Claudia Seardine

- Select once which family to use:

```
WidgetFactory* wf;  
    switch (lf) {  
        case WIN: wf = new WinWidgetFactory();  
            break;  
        case MOTIF: wf = new MotifWidgetFactory();  
            break;  
        ...  
    }
```

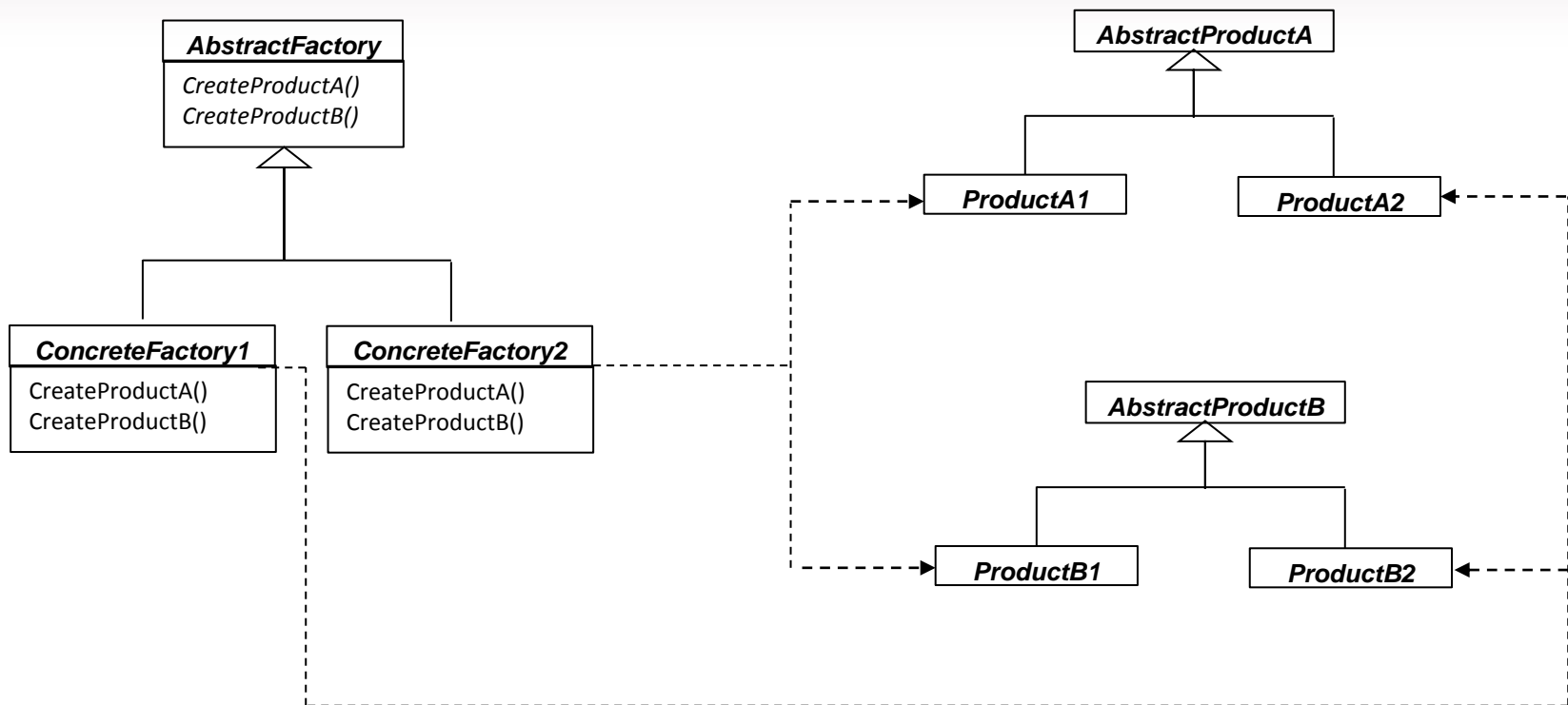
- When creating objects in the code, don't use “new” but call:

```
Button* b = wf->makeButton(args);
```

- Switch families – once in the code
- Add a family – one new factory, no effect on existing code

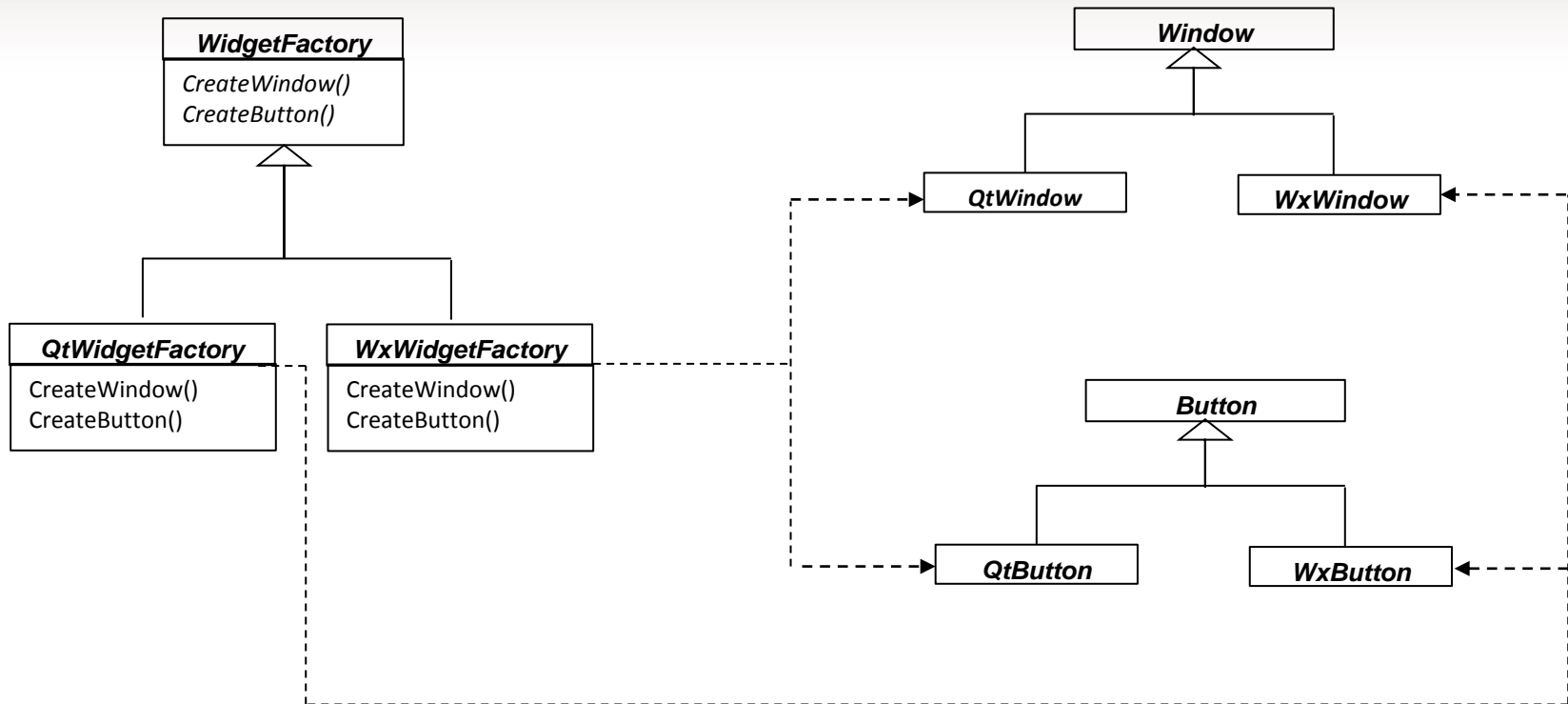
Abstract Factory: diagram

Claudia Scardine



Abstract Factory applied

Claudia Scardine



Comments on Abstract Factory

Claudia Scardine

- Pros:
 - **It makes exchanging product families easy.** It is easy to change the concrete factory that an application uses. It can use different product configurations simply by changing the concrete factory.
 - **It promotes consistency among products.** When product objects in a family are designed to work together, it's important that an application uses objects from only one family at a time.
- Cons:
 - Not easy to extend the abstract factory's interface
- Relation with other patterns:
 - A concrete factory is often a Singleton

Creational patterns:

Factory Method

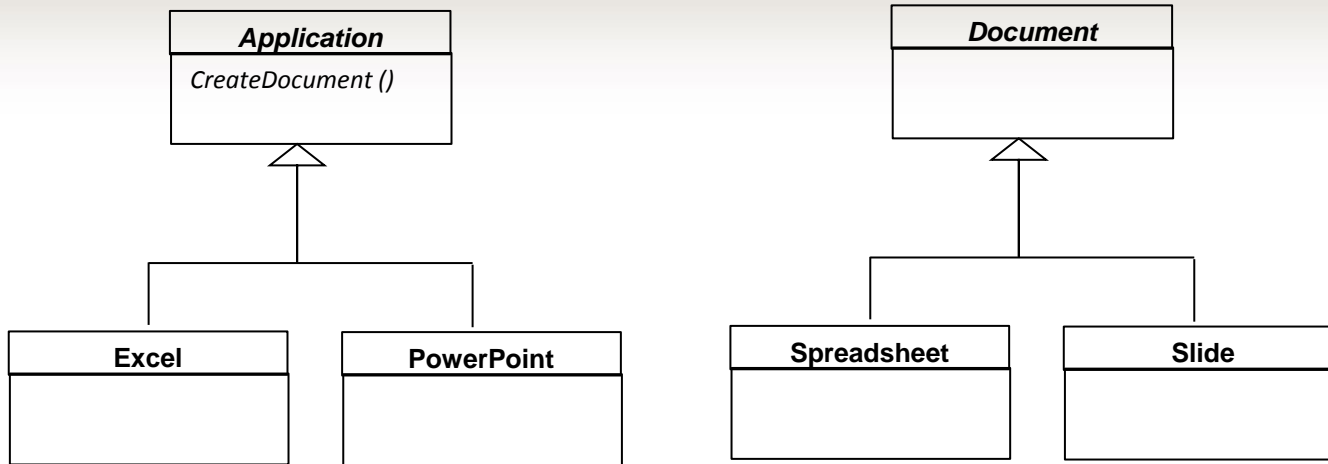
Factory Method: motivation

Claudia Seardine

- We have multiple hierarchies of classes, and the specific subclass of one hierarchy depends on the specific subclass of the other hierarchy
- Example:
 - A framework that handles multiple applications and multiple documents
 - Abstract classes: Application and Document
 - Clients have to subclass these two abstract classes

Factory Method: motivation

Claudia Seardine



- The Application class:
 - Knows *when* a new document should be created
 - Doesn't know *what kind* of document (because it is application-specific)

Solution: Factory Method*

Claudia Seardine

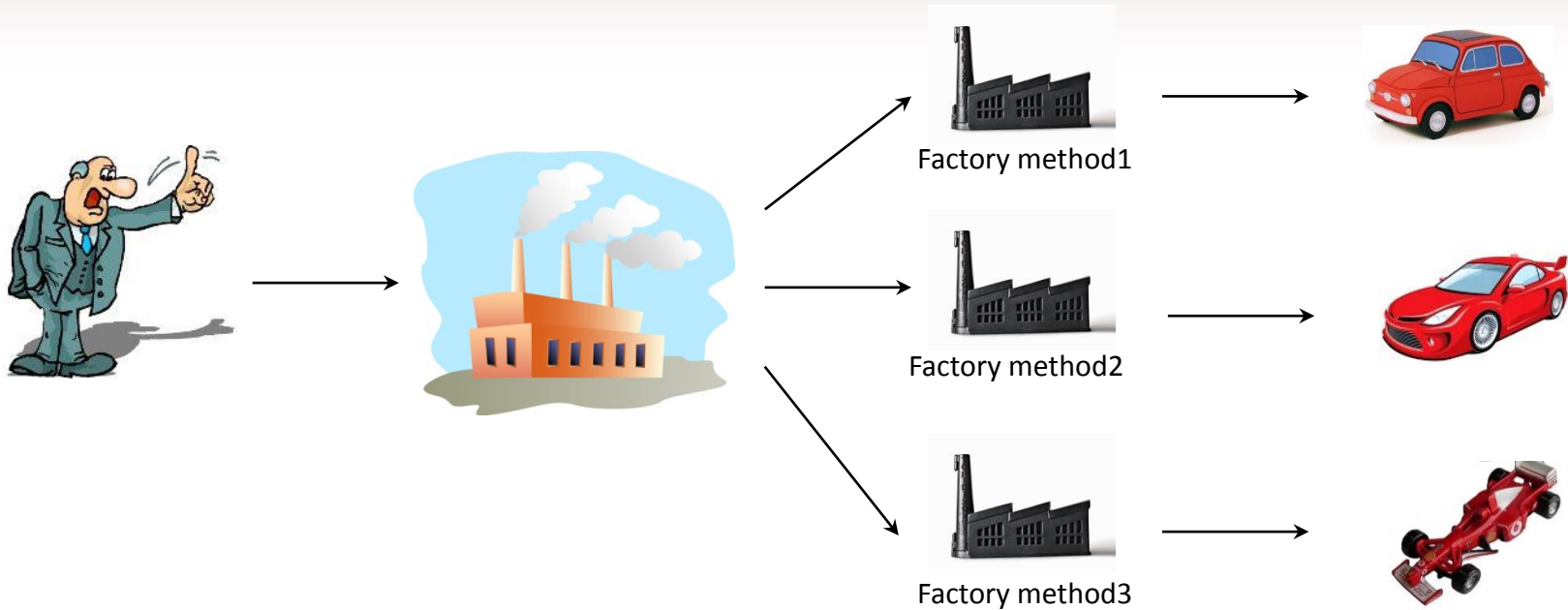
Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory method lets a class defer instantiation to subclasses.

* AKA Virtual Constructor

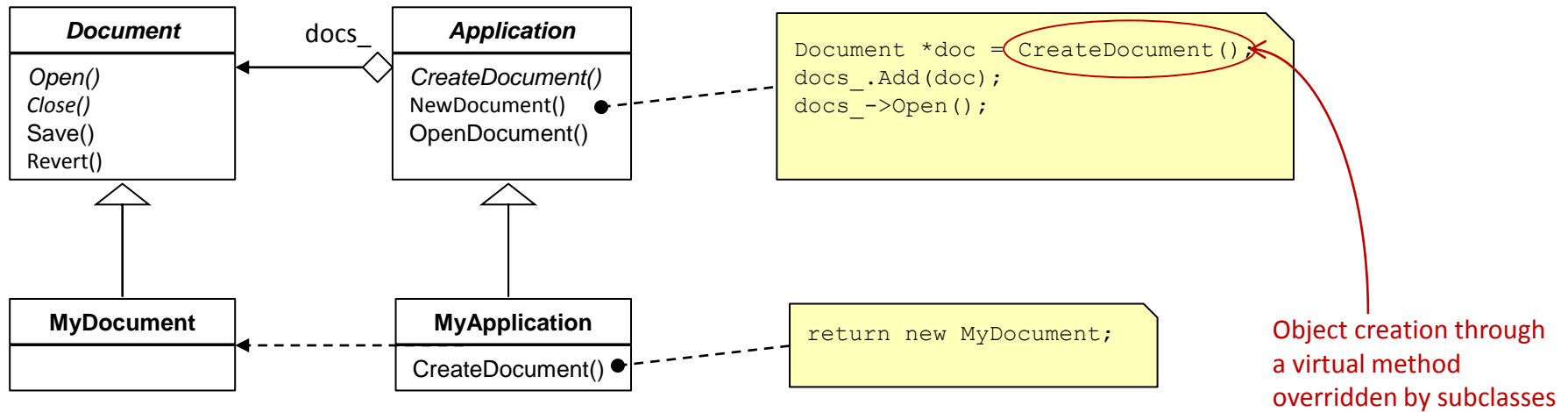
Solution: Factory Method

Claudia Scardine



Solution: Factory Method

Claudia Seardine



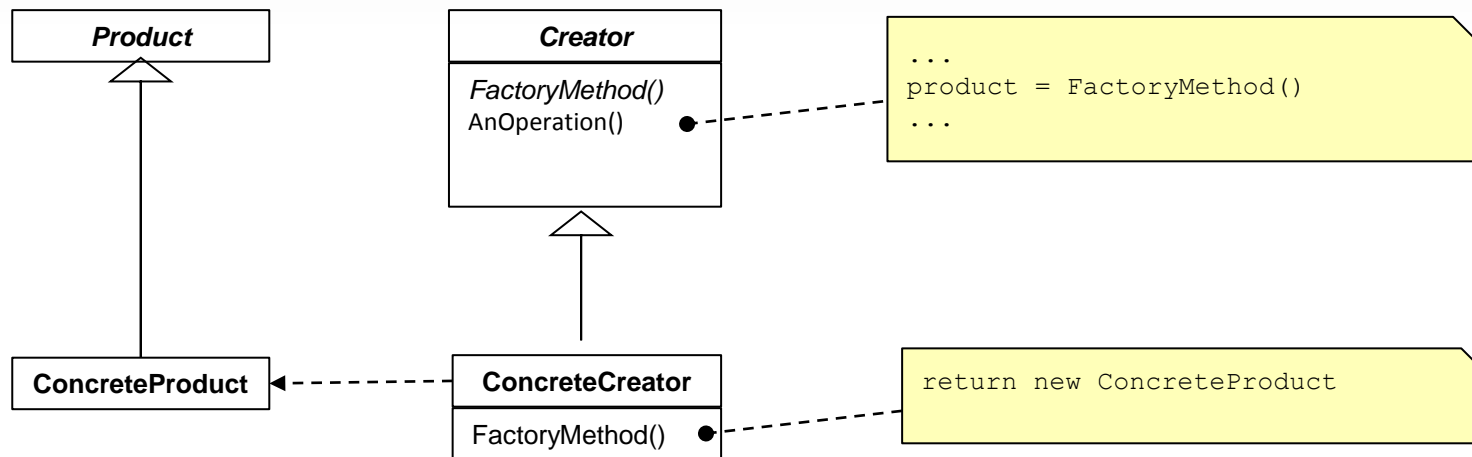
Factory Method: applicability

Claudia Seardine

- Use Factory Method when:
 - A class can't anticipate the class of objects it must create
 - A class wants its subclasses to specify the objects it creates

Factory Method: diagram

Claudia Scardine



Factory Method: participants

Claudia Scardine

- **Product** (Document):
 - Defines the interface of objects the factory method creates
- **ConcreteProduct** (MyDocument)
 - Implements the Product interface
- **Creator** (Application)
 - Declares the factory method, which returns an object of type Product
 - May call the factory method to create a Product object
- **ConcreteCreator** (MyApplication)
 - Overrides the factory method to return an instance of a CConcreteProduct

Factory Method: pros & cons

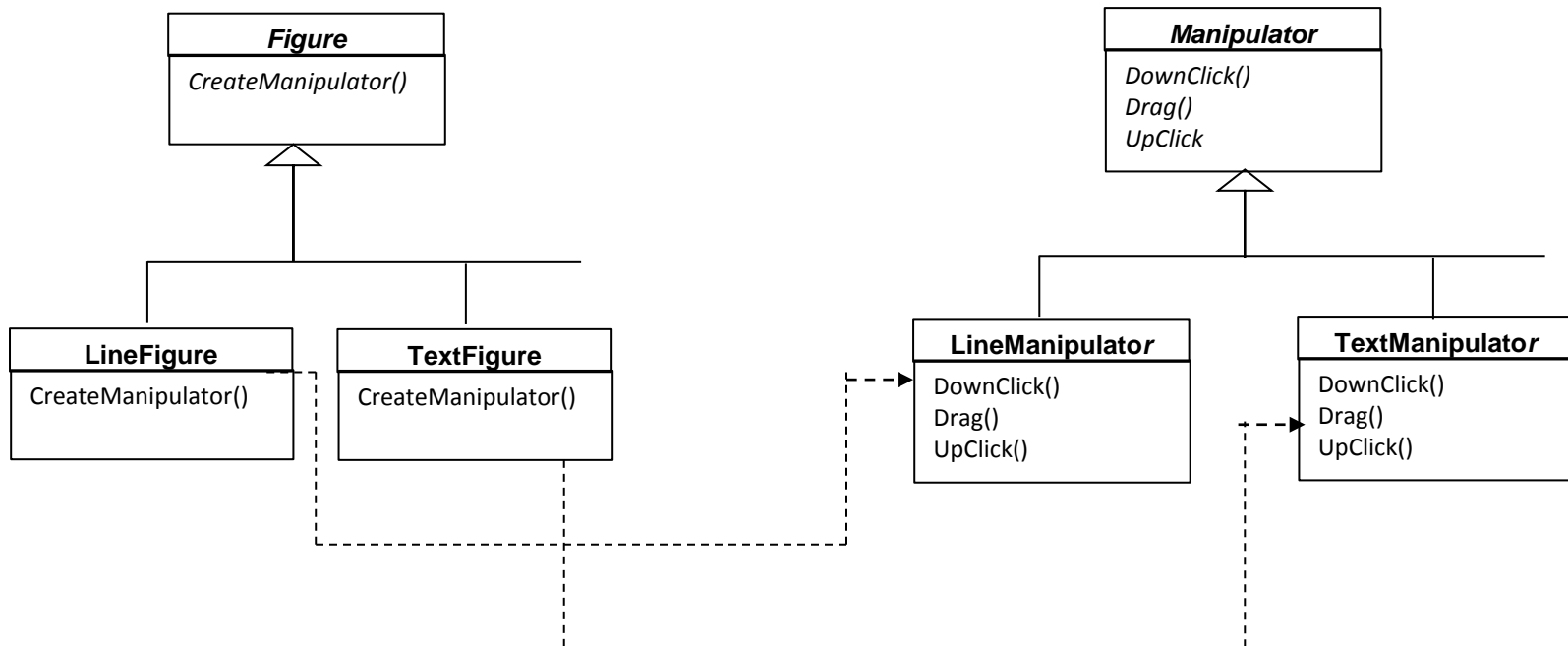
Claudia Seardine

- **Pros:** always more flexible than creating objects directly
- **Cons:** clients have to subclass the Creator to create a particular object

Factory Method: pros & cons

Claudia Seardine

- Factory method allows to connect parallel class hierarchies
- Creation is not the only operation that can be delegated
- E.g. graphical figures can delegate manipulation (stretching, moving, rotation) to other classes which also keep the state of manipulation



Creational patterns: Static Factory Method

Static Factory Method: motivation

Claudia Seardine

- Sometimes the constructor is not as flexible as we wish
 - Cannot handle a pool of resources and reuse existing resources
 - No descriptive names
 - All constructors have the same name
 - Readability of overloaded constructors is poor

```
class NutritionFacts {  
public:  
    NutritionFacts(int servingSize, int servings);  
    NutritionFacts(int servingSize, int servings, int calories);  
    NutritionFacts(int servingSize, int servings, int calories,  
                   int fat);  
    NutritionFacts(int servingSize, int servings, int calories,  
                   int fat, int sodium) {...}  
};  
  
NutritionFacts label1(240, 8, 100, 0, 35, 27);  
NutritionFacts label3(300, 10, 100, 0, 42, 25);
```

Solution: Static Factory method

Claudio Scardine

```
class MyClass {  
public:  
    MyClass (int param);           // std constructor  
    static MyClass *create (int param) { // static factory method  
        return new MyClass(param);  
    }  
};
```

- It has almost nothing to do with GoF's factory method
- Similar to Singleton but
 - No limit on the maximum number of instances
 - Constructor is not kept private, to let subclassing

Structural patterns

Structural patterns

Claudia Seardine

- **Adapter**: convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge**: decouple an abstraction from its implementation so that the two can vary independently. The intent behind Bridge is to allow separate class hierarchies to work together even if they evolve independently.
- **Composite**: compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Decorator** (AKA Wrapper): attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. The enclosing object is called a "*decorator*" and conforms to the interface of the enclosed object so that its presence is transparent to the client.

Structural patterns (2)

Claudia Seardine

- **Facade:** provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Flyweight:** use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy** (AKA Surrogate): provide a surrogate or placeholder for another object to control access to it or to defer its creation to when it's really needed (i.e., "*on demand*"). Examples: smart pointers, copy-on-write.

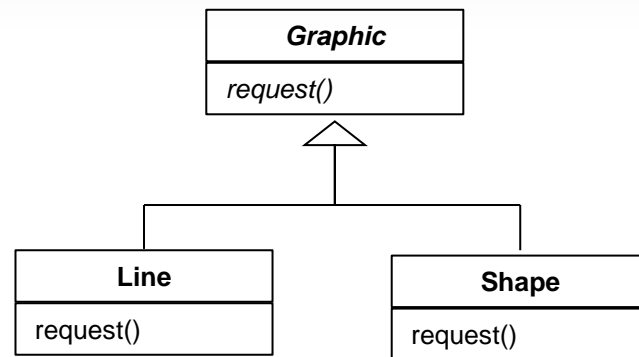
Structural patterns:

Adapter

Adapter: motivation

Claudia Scardine

- We have implemented a toolkit to draw graphics objects
 - The base class is `Graphic`, then we have `CompositeGraphic`, and `Line`, `Shape`, etc.



- We would like to add a `TextBox`, but it is difficult to implement it
- So we decide to buy an off-the-shel library that implements `TextBox`
- However, `TextBox` does not derive from `Graphic`.
 - We don't have their source code
 - How to include it in our framework?

Solution: Adapter

Claudia Seardine

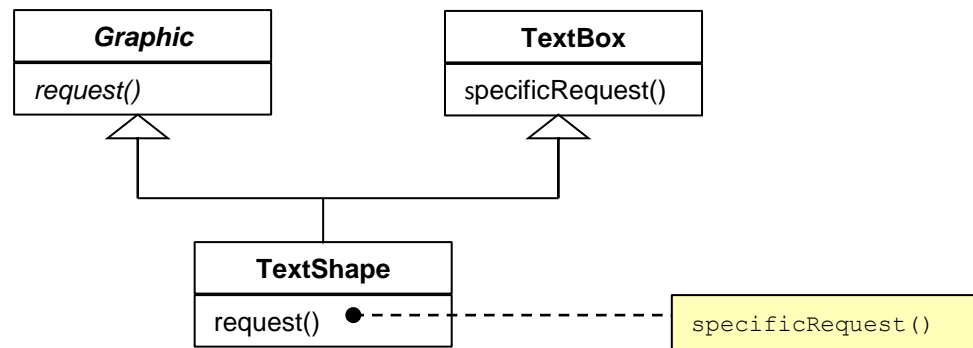
Convert the interface of a class into another interface clients expect.

Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

First option: class adapter

Claudia Seardine

- Use multiple inheritance
- Implement a `TextShape` class that derives from `Graphic` and from `TextBox`, reimplementing part of the interface:

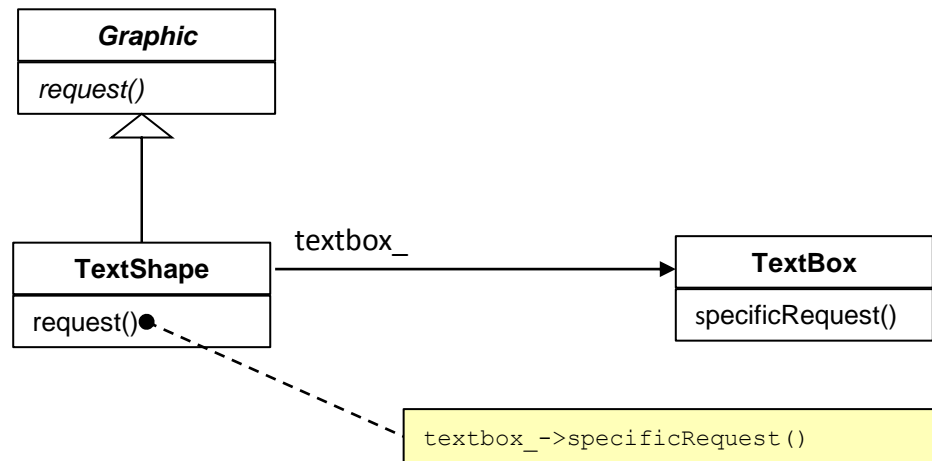


- Expose interfaces of the adaptee (`TextBox`) to clients
- Allow the adapter (`TextShape`) to be used in places where an adaptee is expected

Second option: object adapter

Claudia Seardine

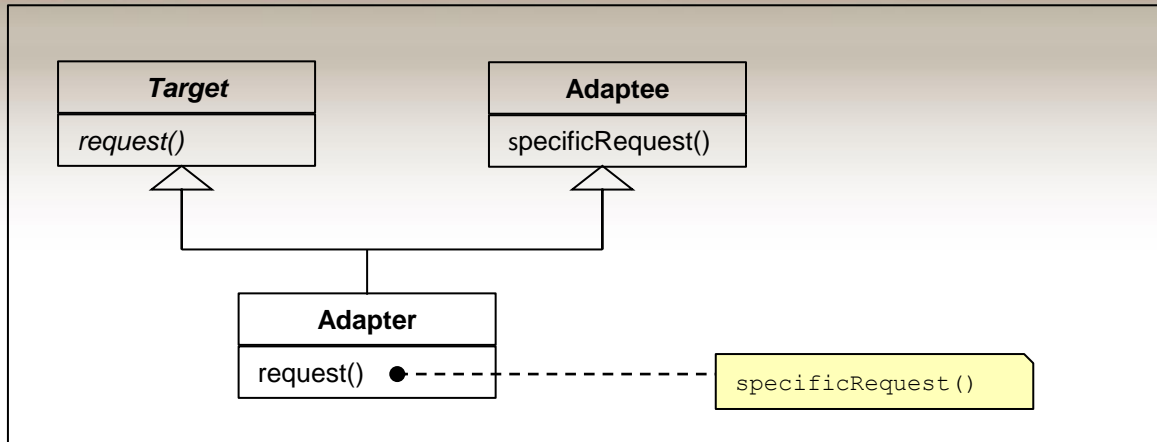
- Use composition
- Implement a `TextShape` class that derives from `Graphic`, and holds an object of type `TextBox` inside:



- Does not expose interfaces of the adaptee (`TextBox`) to clients (safer)
- Where an adaptee (`TextBox`) is expected, the adapter (`TextShape`) needs to provide a function `getAdaptee()` to return the pointer to the adaptee object.

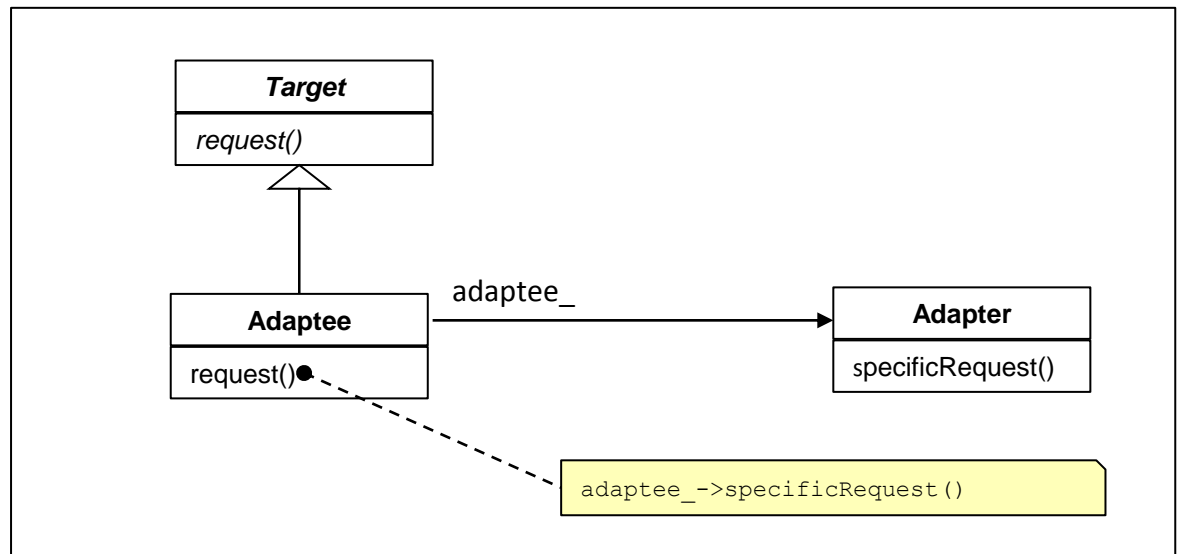
Adapter: diagrams

Claudia Scardine



Class Adapter

Object Adapter



Adapter: another example

Claudia Seardine

- Suppose we are given the class:

```
class DocManager {  
public:  
    void printDocument();  
    void saveDocument();  
};
```

which is responsible for opening, closing, saving, and printing text documents

- Now suppose that we want to build a word-processor:
 - GUI that allows – among other things – to save file clicking on Save button, and print by clicking on Print button.
 - Details of saving and printing handled by DocManager

Adapter: another example

Claudia Scardine

- Observe: two instances and one link



- Question: how to design Button so that it can collaborate with DocManager ?

Adapter: another example

Claudia Scardine

- Possible design:

```
class Button {  
    protected:  
        DocManager**target_;  
        void buttonPressed() {  
            target_>printDocument();  
        }  
};
```

To invoke printDocument() Button needed to know the class (DocManager) of the target object.



However:

- Button should not care that target is a DocManager
- It's not requesting information: it's just saying *"Hey, I've been pressed. Do something!"*

Adapter: another example

Claudia Scardine

- Typical solution: use an interface
 - We define an interface ButtonListener
 - The interface declares the events that an object must handle to collaborate with a Button
 - We let DocManager inherit from this interface

Adapter: another example

Claudia Seardine

```
class Button {  
public:  
    Button(const std::string& lab) : label_(lab), target_(0) {}  
    void setListener(ButtonListener* blis) {  
        target_ = blis;  
    }  
protected:  
    std::string label_;  
    ButtonListener* target_;  
    void buttonPressed() {  
        if (target_)  
            target_->buttonPressed(label_);  
    }  
};
```

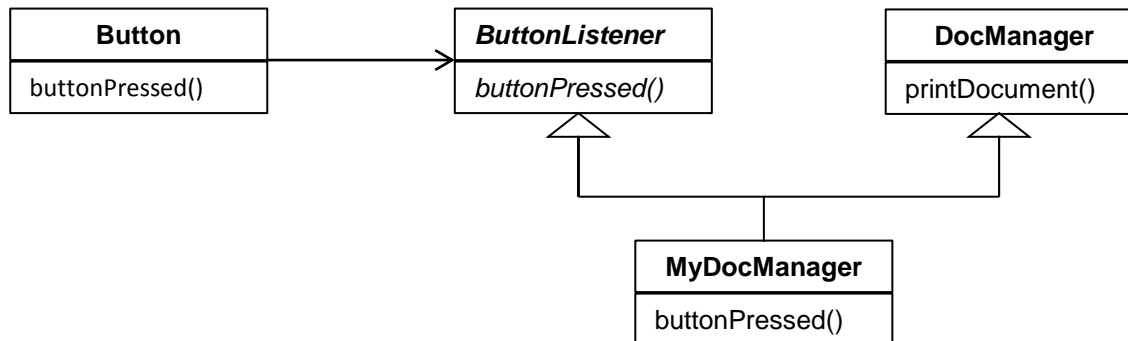
```
class ButtonListener {  
public:  
    virtual void buttonPressed  
        (const std::string&) = 0;  
};
```

```
class DocManager: public ButtonListener {  
    //...  
};
```

Adapter: another example

Claudia Scardine

- What if DocManager is already implemented and tested and does not inherit from this interface ?
- We can use the adapter pattern:



Structural patterns: Composite

Composite: motivation

Claudia Seardine

- We must write a complex program that has to treat several objects in a hierarchical way
 - Objects are composed together to create complex objects
 - Composite objects must be treated like simple ones
 - E.g. a painter handling shapes that can be composed of simpler shapes (lines, squares, etc.)
 - E.g. a word processor which allows users to compose pages consisting of letters, figures, and other objects

Composite: motivation

Claudia Seardine

- Requirements:
 - Treat simple and complex objects uniformly in code – move, erase, rotate and set color work on all
 - Composite objects can be made of other composite objects

Solution: Composite

Claudia Scardine

*Compose objects into tree structures to represent part-whole hierarchies.
Composite lets clients treat individual objects and compositions of objects uniformly.*

Solution: Composite

Claudia Scardine

- All simple objects inherit from a common interface, say Graphic:

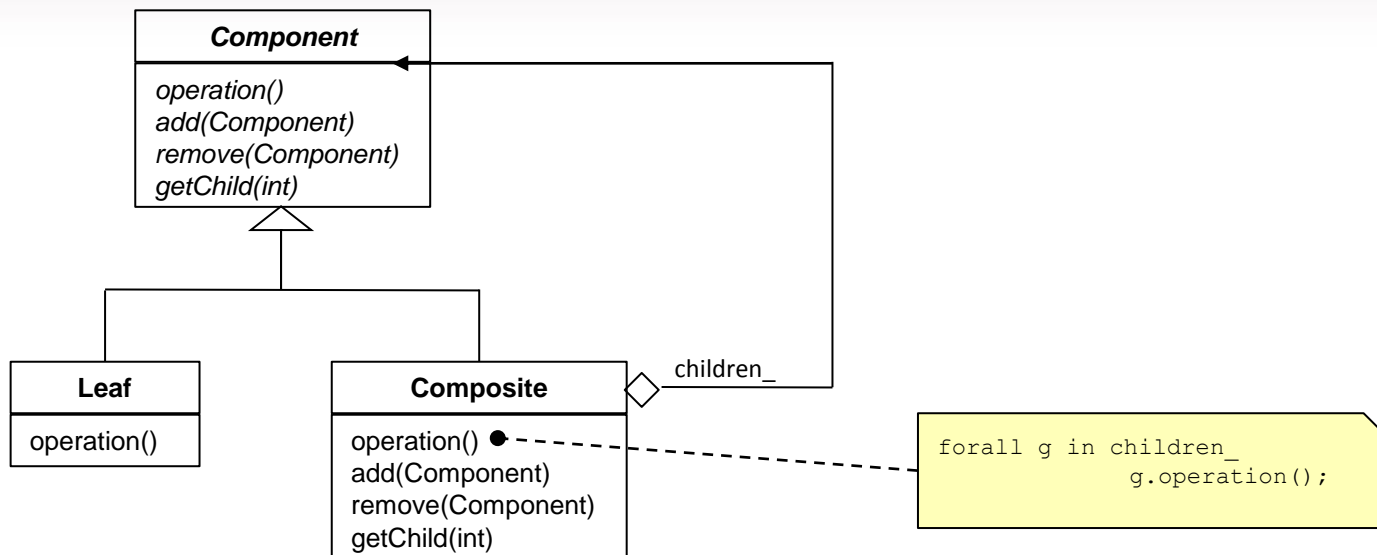
```
class Graphic {  
public:  
    virtual void move(int x, int y) = 0;  
    virtual void setColor(Color c) = 0;  
    virtual void rotate(double angle) = 0;  
};
```

- The classes Line, Circle and others inherit Graphic and add specific features (radius, length, etc.):

```
class CompositeGraphic : public Graphic {  
public:  
    void rotate(double angle) {  
        for (int i=0; i < count(); ++i)  
            items_[i].rotate(angle);  
    }  
private:  
    std::list<Graphic> items_;  
};
```

Composite: diagram

Claudia Scardine



Composite: participants

Claudia Scardine

Component (Graphic)

- Declares the interface for objects in the composition
- Implements default behaviour for the interface common to all classes
- Declares an interface for accessing and managing its child components
- (optional) defines an interface for accessing a component's parent

Leaf (Rectangle, Line, Text, etc.)

- Represents leaf objects in the composition. A leaf has no children.
- Defines behaviour for primitive objects in the composition

Composite (Picture)

- Defines behaviour for components having children
- implements child-related operations in the Component interface

Client

- manipulates objects in the composition through the Component interface

Trade-off: transparency vs safety

Claudia Seardine

- Important issue: which class declares the add/remove operations for children management ?
- For **transparency**:
 - In Component, at the root of the hierarchy
 - All components are treated uniformly
 - It costs safety, because clients may do silly things like add/remove objects from leaves
- For **safety**:
 - In the Composite class
 - Any attempt to add/remove objects from leaves will be caught at compile-time.
 - Less transparency, because leaves and composites have different interfaces

Structural patterns:

Decorator

Decorator: motivation

Claudia Seardine

- Sometimes we need to add responsibilities to individual objects (rather than to entire classes)
 - E.g., in a GUI, add borders and scrollbars to a widget
- *Inheritance* approach:
 - We could let the class `Widget` inherit from a `Border` class
 - Not flexible, because for any additional responsibility, we must add additional inheritance
 - Not flexible because it is static: the client cannot control dynamically whether to add or not add the border
- *Composition* approach:
 - Enclose the component into another object which additionally implements the new responsibility (i.e., draws the border)
 - The enclosing object is called **decorator** (or wrapper) and has the same interface of the enclosed object.
 - More flexible: it can be done dynamically

Decorator (AKA Wrapper) pattern

Claudia Seardine

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The enclosing object is called a "decorator" and conforms to the interface of the enclosed object so that its presence is transparent to the client.

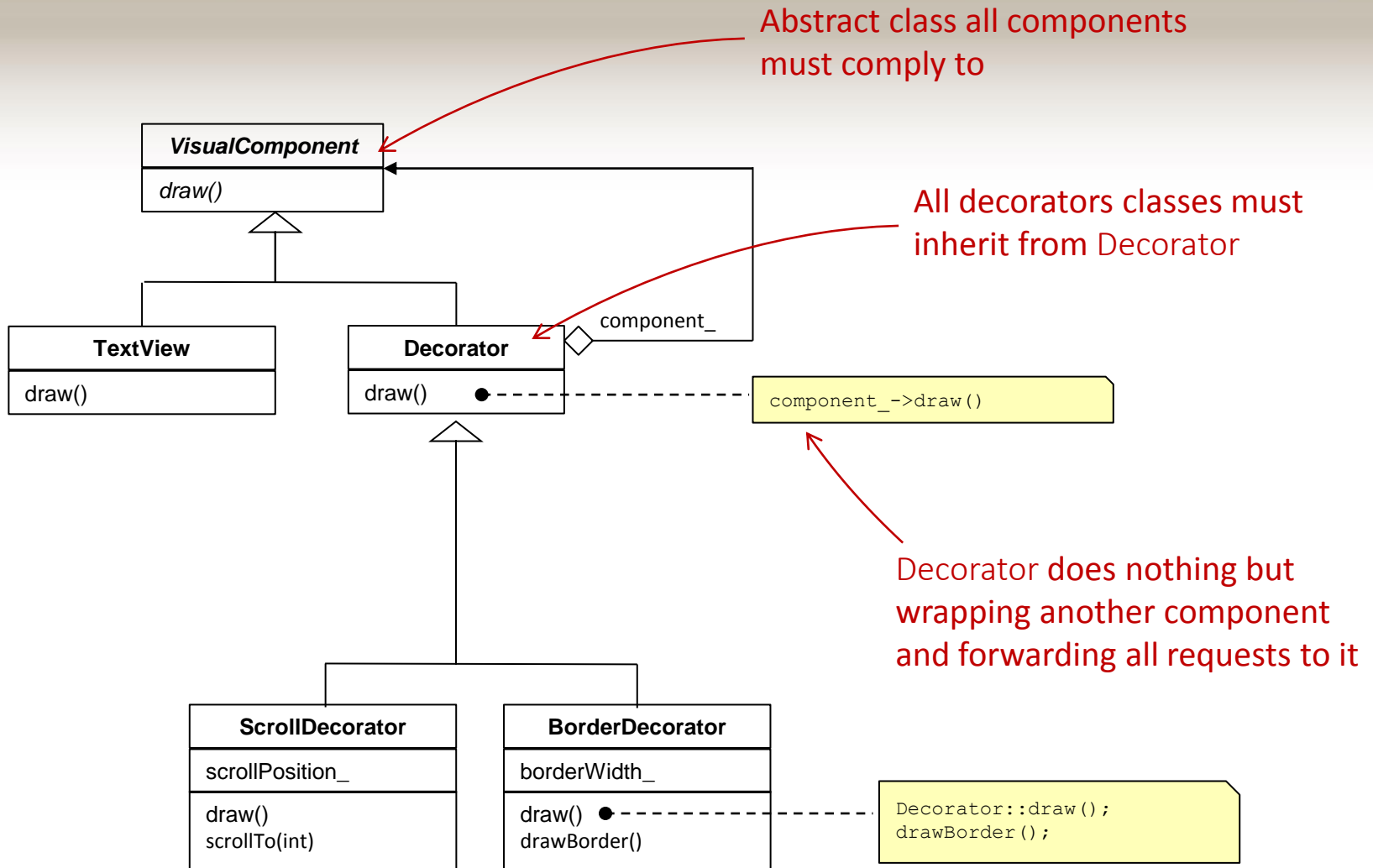
Decorator: example

Claudia Scardine

- A TextView object displays text in a window
- TextView class has no scrollbars by default
- We add a ScrollDecorator to attach scrollbars to a TextView object
- What if we also want a border ? We also attach a BorderDecorator!

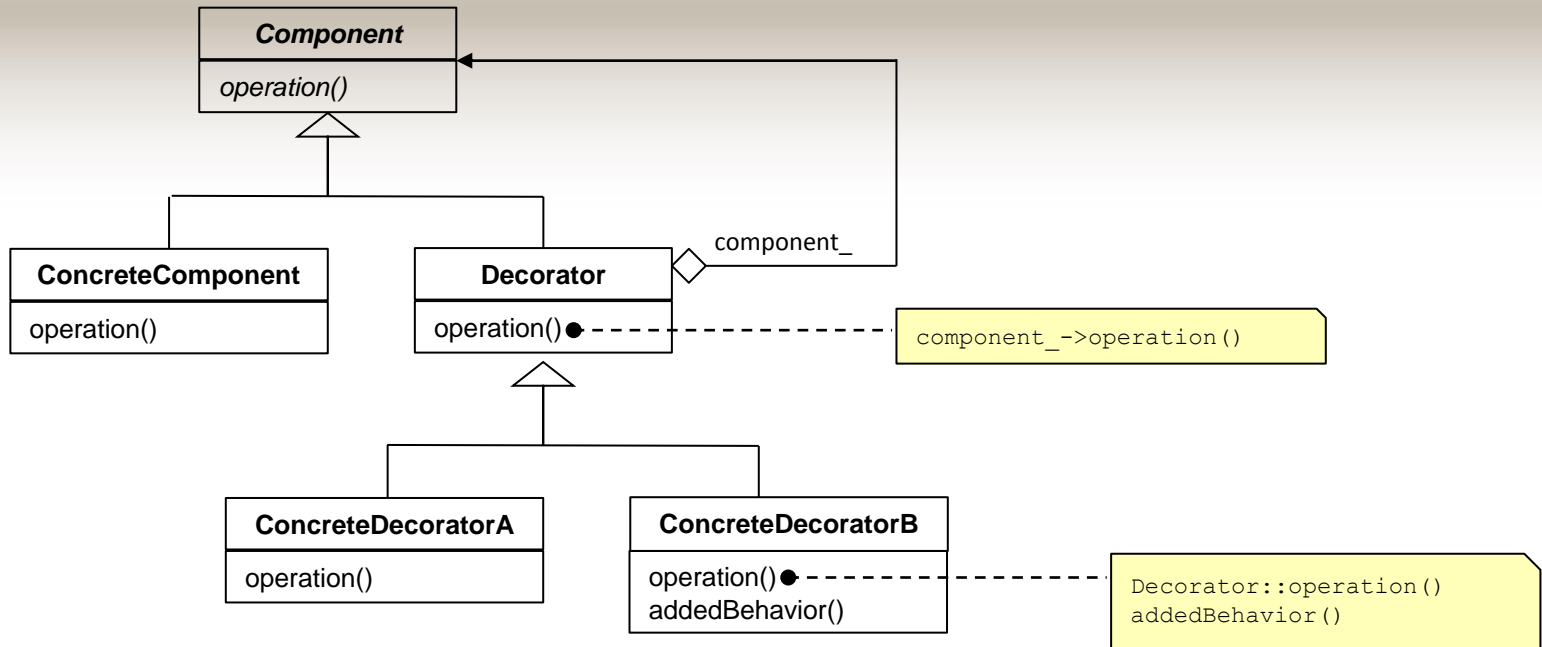
Decorator: example

Claudia Scardine



Decorator: diagram

Claudia Scardine



- Key properties:
 - any decorator implements the same interface of the component
 - therefore, from the client point of view, it does not matter if an object is decorated or not

Decorator: notes

Claudia Scardine

- When to use
 - To add responsibilities to individual objects dynamically and transparently
 - When extension by sub-classing is impractical
- Don't use if the component class has large interface
 - Implementation becomes too heavy (all methods must be re-implemented by forwarding to the internal component)
- When there are many “decoration options” you may end up with a lot of small classes
 - The library may become more complex to understand and to use due to the large number of classes
- Where it is used:
 - In the Java I/O library, the most basic classes perform raw input/output



Structural patterns:

Proxy

Proxy: motivation

Claudia Scardine

- Sometimes it is impossible, or too expensive, to directly access an object
 - The object could be physically on another PC (in distributed systems)
- Some other times we just want to control the access to an object
 - Example: smart pointers
- In other cases, it is too expensive to create the object, so it should be created *"on demand"*
 - Example: copy-on-write
- In such cases, we could use a placeholder for the object, that is called *Proxy*
 - AKA Surrogate

Provide a surrogate or placeholder for another object to control access to it or to defer its creation to when it's really needed (i.e., "on demand").

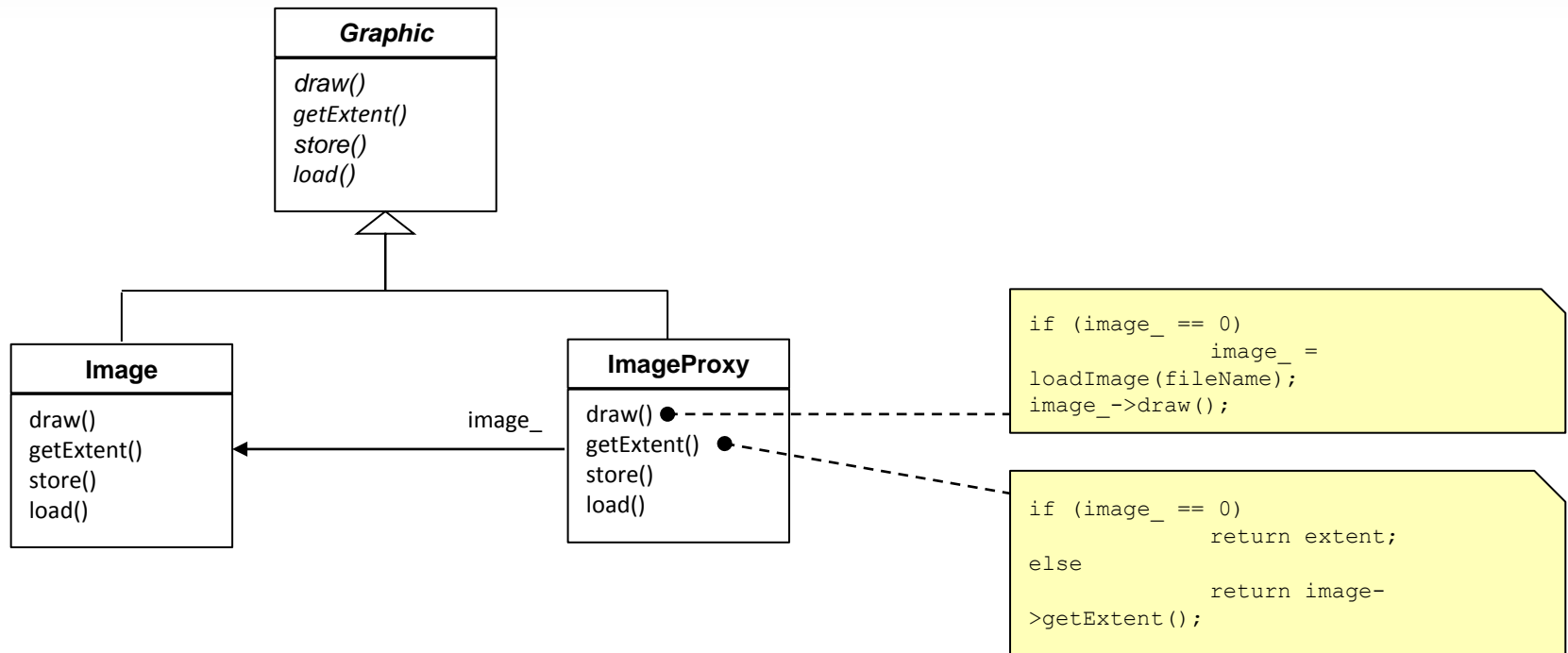
Proxy: example

Claudia Scardine

- Suppose that a text document can embed images
- Some of these can be very large, and so expensive to create and visualize
- On the other hand the user wants to load the document as soon as possible
- The solution could be to use a “similar object” that has the same methods as the original image objects
- The object will start loading and visualizing the image only when necessary (for example, on request for `draw()`), and in the meanwhile could draw a simple white box in its placeholder
- It can also return the size of the image (height and width) that it is what is needed to format the rest of the document

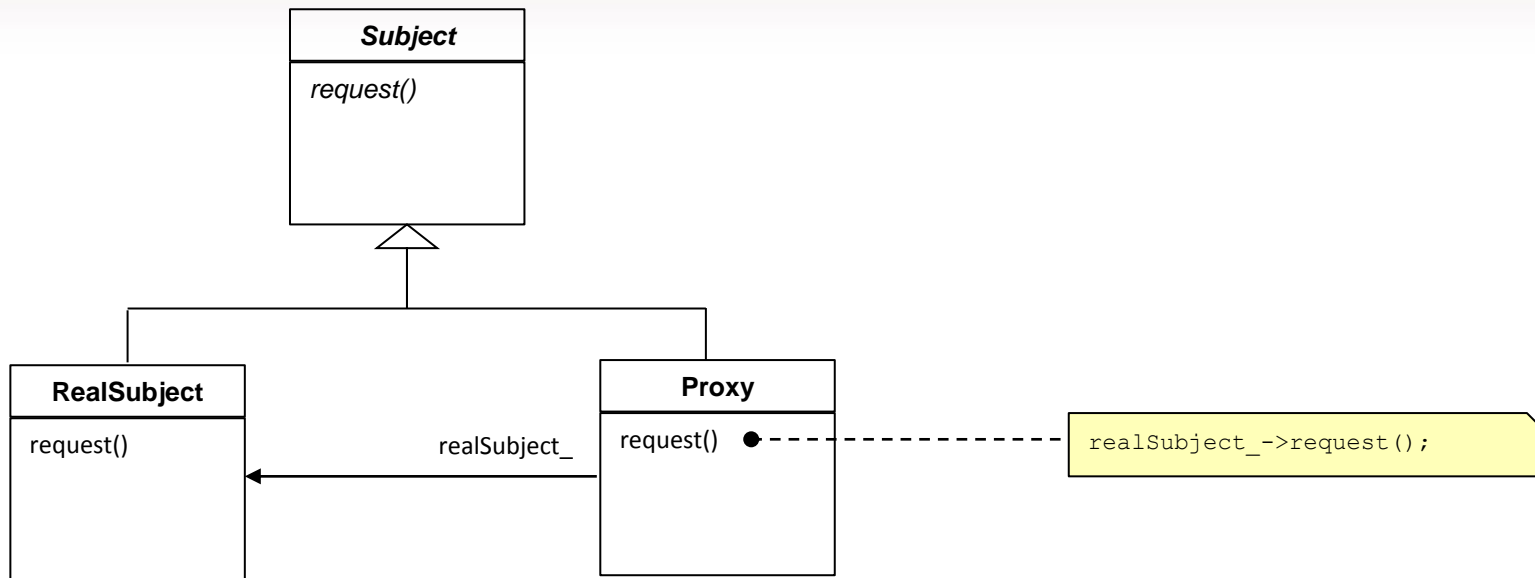
Proxy: example

Claudia Scardine



Proxy: diagram

Claudia Scardine



Proxy: participants

Claudia Scardine

- **Proxy** (`ImageProxy`)
 - Maintains a reference that lets the proxy access the real subject. Its interface must be the same as the one of the `RealSubject`.
 - Provides an interface identical to `Subject`'s so that a proxy can be substituted for the real subject
 - Controls access to the real subject and may be responsible for creating and deleting it
 - Other responsibilities depends on the kind of proxy
- **Subject** (`Graphic`)
 - Defines the common interface for `RealSubject` and `Proxy` so that a proxy can be used anywhere a `RealSubject` is expected
- **RealSubject** (`Image`)
 - Defines the real object that the proxy represents

Proxy: applicability

Claudia Seardine

- Proxy is applicable whenever there is the need for a more sophisticated reference to an object that a simple pointer
 - A **remote proxy** provides a local representative for an object in a different address space
 - A **virtual proxy** creates an expensive object on demand (like in the example)
 - A **protection proxy** controls access to the original object.
 - Protection proxies are useful when objects should have different access rights
 - A **smart reference** is a replacement for a bare pointer (see *shared_ptr*)
 - Implement *copy-on-write* behaviour when copying large objects

Structural patterns: comparison

Claudia Seardine

- Many of the patterns we have seen have similar class diagrams
- Basic idea:
 - Favour composition for extending functionality,
 - Use inheritance to make interfaces uniform
- However, they differ in the intent

Structural patterns: comparison (2)

Claudia Seardine

- Composite vs Decorator:
 - Similar recursive structure
 - Composite used to group objects and treat them as one single object
 - Decorator used to add functionalities to classes without using inheritance
 - These intents are complementary therefore, these patterns are sometimes used in concert
- Decorator vs Proxy:
 - Both provide the same interface as the original object
 - However:
 - Decorator is used to add functionalities
 - Decorators can be wrapped one inside the other
 - Proxy is used to control access

Behavioral patterns

Behavioral patterns

Claudia Scardine

- **Chain of Responsibility:** avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- **Command:** Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
- **Interpreter:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Iterator:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator:** Define an object that encapsulates how a set of objects interact, Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Behavioral patterns (2)

Claudia Scardine

- **Memento:** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- **Observer** (AKA Publish-Subscribe): Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **State**: allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- **Strategy** (AKA Policy): define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Template Method:** define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- **Visitor**: represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Behavioral patterns: Observer

Observer: motivation

Claudia Seardine

- We need to maintain consistency among (weakly-)related object
 - When something happens to an object, other objects must be informed
- Typical example in GUIs
 - The Document object must be informed when a button “Print” is clicked, so that the print() operation can be invoked
 - The ViewPort object must be informed when the window is resized(), so that it can adjust the visualization of the objects
- Participants:
 - An observed object ("subject")
 - Another object that wants to be informed ("observer")

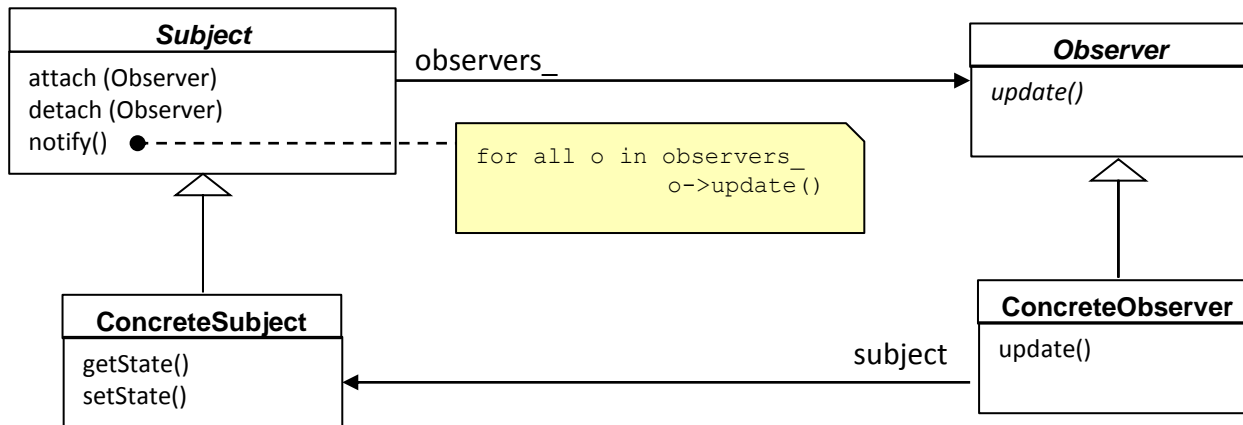
Observer pattern

Claudia Seardine

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- AKA "Publish-Subscribe"
- Participants:
 - An observed object (subject)
 - Another object that wants to be informed (observer)

Observer: diagram



Observer: example

Claudia Scardine

- The user resizes a window:
 - Every component of the window needs to be informed of a resize operation (viewport, scrollbars, toolbars, etc.)
 - In this way, every object can synchronize its state with the new window size
- Solution:
 - The window can install *observers*
 - All components (viewport, scrollbar, etc.) can attach an observer to the main window that is informed when a resize operation is under way
 - The observer asks for the current size of the window, and invoke methods on the objects to adjust their state (size)

Observer: pros

Claudia Scardine

- **Abstract coupling** between subject and observer
 - All that a subject knows is that there is a list of observers, but it does not know anything about the observers themselves
 - The observer instead must know the subjects
- **Broadcast communication**
 - There can be many independent observers, with different purposes and hierarchies
 - Example: resizing a window can affect the viewports inside the window, the scrollbars, etc.

Observer: cons

Claudia Scardine

- **Unexpected updates**
 - A seemingly innocuous operation on the subject may cause a cascade of updates on the observers and their dependent objects, many of them may not care about any update
 - This simple protocol does not tell the observer *which* change happened to the subject (a resize? a move?)

Observer: update models

Claudia Scardine

- **Pull model** (what we have already seen)
 - The subject sends nothing
 - The observer asks for details about the changes
- **Push model**
 - The subject sends the observer detailed information about the change
 - The observer can understand if it is interested in the change by analysing this additional parameter
- **Specifying events**
 - By complicating the protocol, it is possible to register to specific changes
 - `onResize()`
 - `onMove()`,
 - ...
 - More efficient, but more complex interface



Behavioral patterns:

State

State

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

- This pattern is useful to implement simple state machines
- The idea is to implement each state with a different class, and each event with a different method

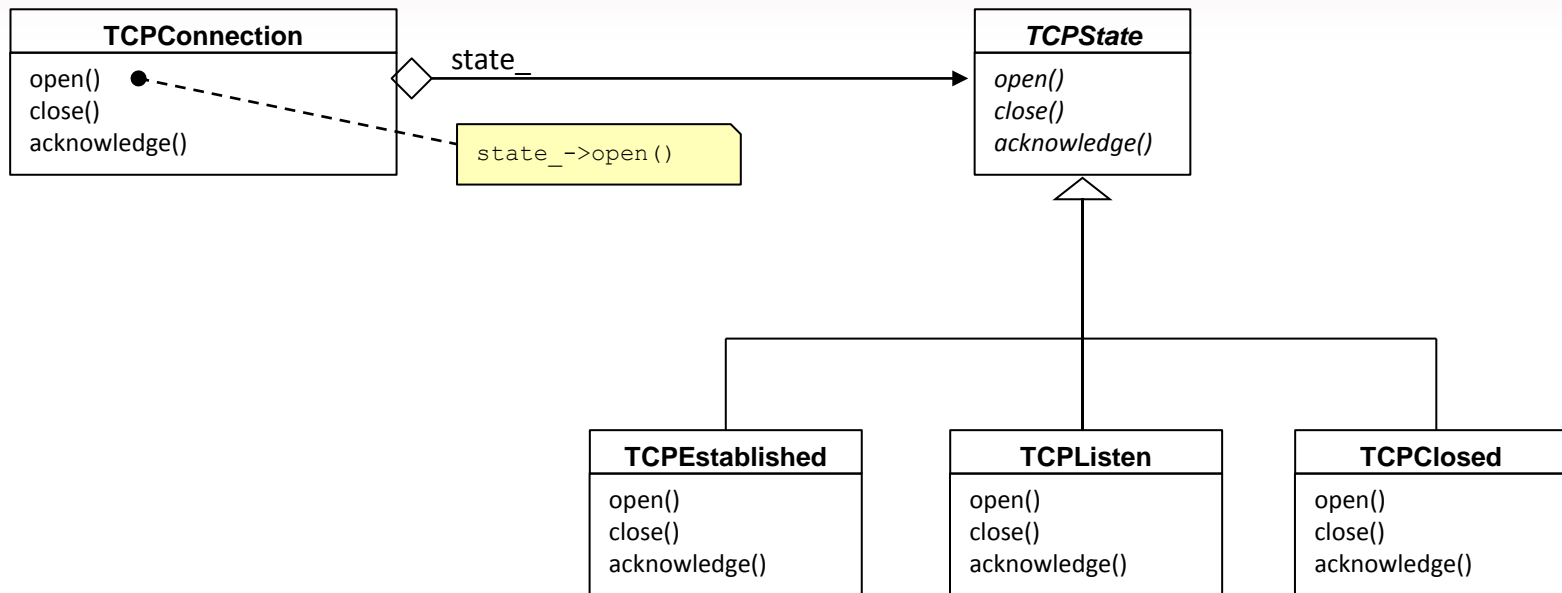
State: example

Claudia Scardine

- Consider a library to implement the TCP protocol
 - A **TCPConnection** can be in one of several different states
 - For example, the connection can be void, established, closing, etc.
 - The response to a request of open depends on the current state of the connection: only if the connection is not yet established we can open it
- This behaviour can be implemented as follows:
 - An abstract class **TCPState** that implements a method for each possible request
 - The derived classes represent the possible states
 - Only some of them will respond to a certain request

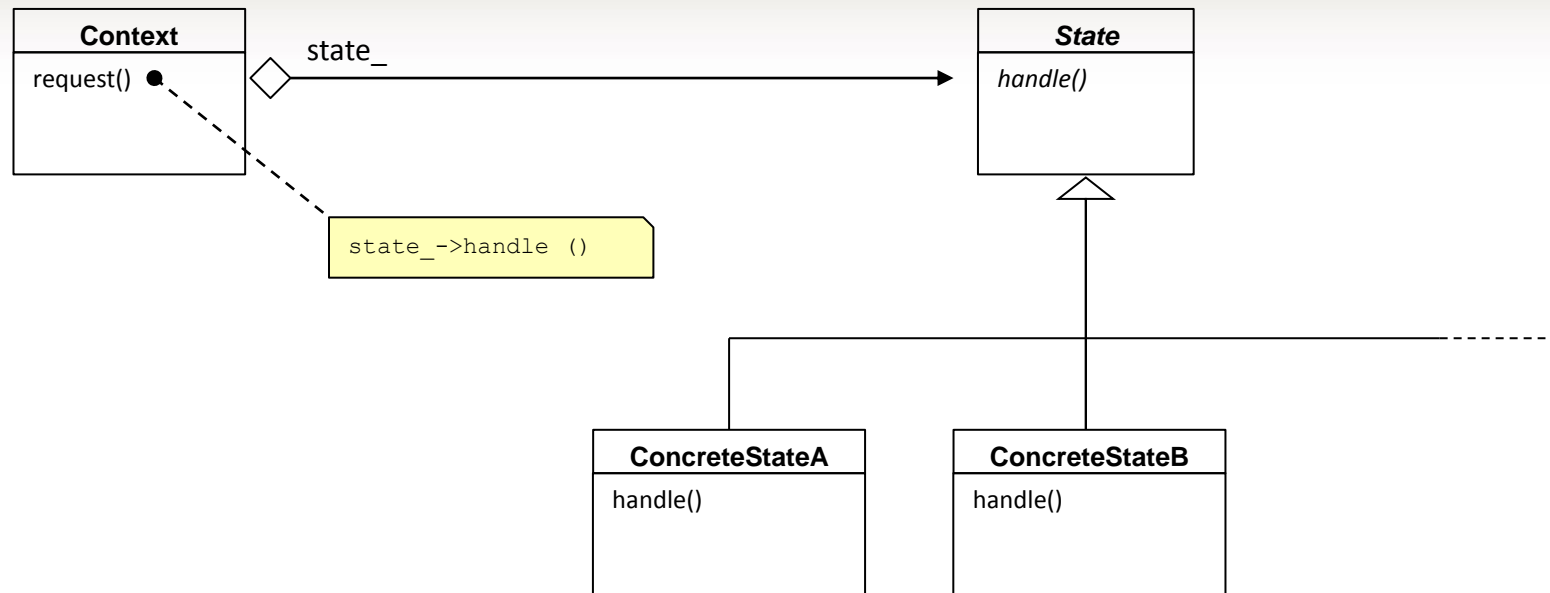
State: example (2)

Claudia Scardine



State: diagram

Claudia Scardine



State: participants

Claudia Scardine

- **Context** (TCPConnection)
 - Defines the interface of interest to clients
 - Maintains an instance of a ConcreteState subclass that defines the current state through a pointer to the abstract State class
- **State** (TCPState)
 - Defines an interface for encapsulating the behaviour associated with a particular state of the Context
- **ConcreteState** subclasses
 - Each subclass implements a behaviour associated with a state of the Context

State: pros

Claudia Seardine

- State-specific behavior is localized and partitioned
 - All behavior associated with a particular state is localized into a single class (ConcreteState)
 - new states and transitions can be easily added
 - the pattern avoids large if/then/else conditionals
- It makes state transition explicit
 - State transitions are atomic
 - State objects are protected from inconsistent internal states

State: cons

Claudia Scardine

- Behavior for different states is scattered across several subclasses
- Higher number of classes
- Code is less compact

State: applicability

Claudia Seardine

- Use the State pattern in one of the following cases
 - An object behaviour depends on its state, that will change at run-time
 - Operations have *large conditional statements* that depend on the object state. This state is usually represented by one or more enumerated constants

Behavioral patterns: Strategy

Strategy: example

Claudia Scardine

- Many algorithms exist for breaking a stream of text into lines
- Hard-wiring all algorithms into the classes isn't desirable for several reasons:
 - Classes get more complex and harder to maintain, especially if they need to support multiple algorithms
 - We don't want to support multiple algorithms if we don't use them
 - Adding new algorithms is difficult and requires to modify existing classes
- We can avoid all the problems by encapsulating the algorithms.
- An algorithm that's encapsulated in this way is called a **strategy**

Strategy pattern*

Claudia Seardine

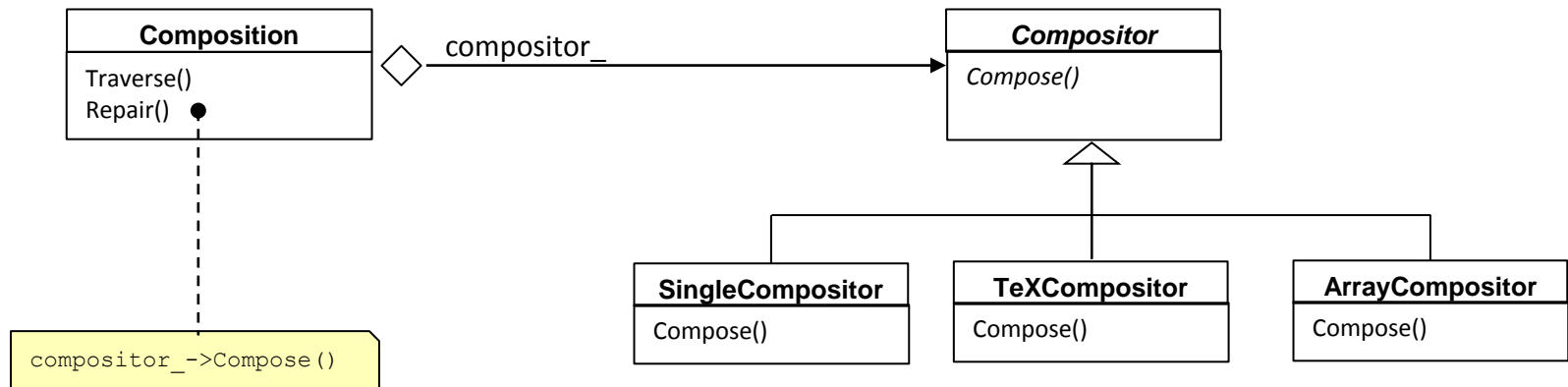
Define a family of algorithms, encapsulate each one, and make them interchangeable.

Strategy lets the algorithm vary independently from clients that use it.

* AKA "Policy"

Strategy: example

Claudia Scardine



Strategy: applicability

Claudia Seardine

Use Strategy when:

- Many related classes differ only in their behavior
- You need different variants of an algorithm
- An algorithm uses data that shouldn't be exposed to the clients
- A class defines many behaviors which appear as multiple conditional statements

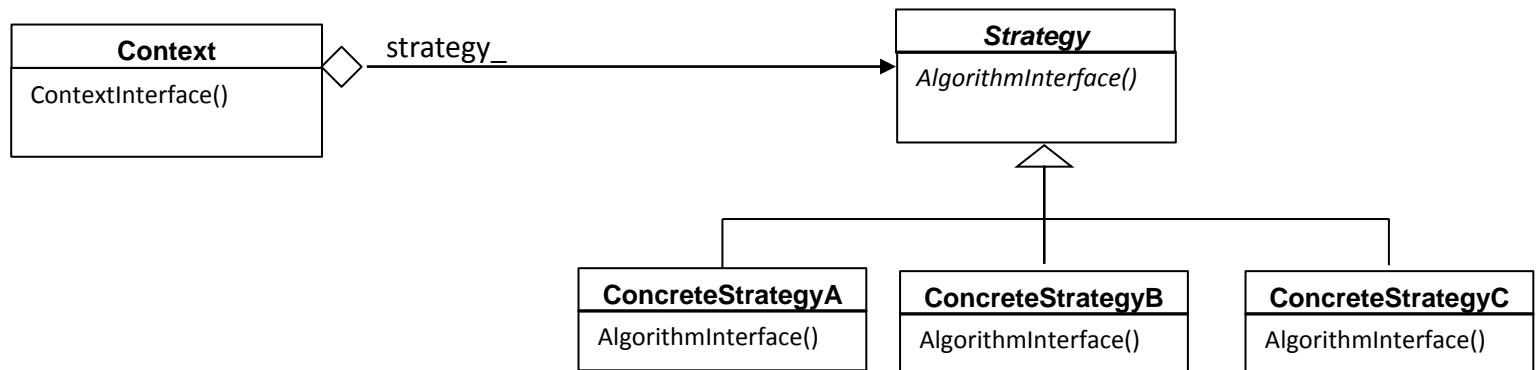
Strategy: notes

Claudia Scardine

- Strategy allows to delegate an algorithm to a class instead of embedding it into the normal code:
 - We make the algorithm general and reusable
 - We can easily change the algorithm by substituting the function
- Why the code is embedded in a class instead of in a function ?
 - Because functions are stateless

Strategy: diagram

Claudia Scardine



Strategy:participants

Claudia Scardine

- **Strategy** (Compositor)
 - Declares the interface common to all supported algorithms
- **ConcreteStrategy** (SimpleCompositor, TexCompositor, ArrayCompositor)
 - Implements the algorithm
- **Context** (Composition)
 - Is configured with a ConcreteStrategy object
 - It maintains a reference to a Strategy object
 - It may define an interface to let Strategy access its data

Strategy: pros & cons

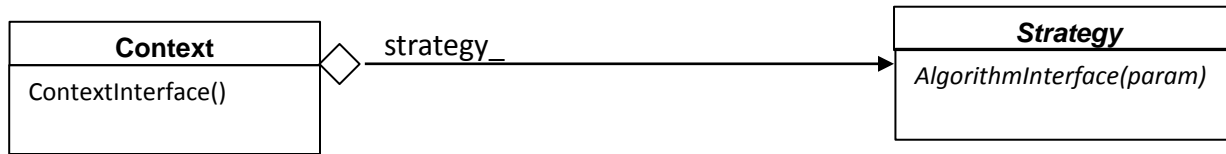
Claudia Scardine

- Pros:
 - Code not exposed to the clients
 - The algorithm can be varied dynamically
 - Conditional statements are eliminated
 - It allows to provide different implementations of the same behavior
- Cons:
 - Communication overhead: all algorithms (also the simple ones) share the same API, which may include unused variables
 - Increased number of objects

Strategy: implementation

Claudia Scardine

- The Strategy and Context interfaces must give access to any needed data
- First option:
 - Context passes data as parameters to Strategy operations:



- This keeps Strategy and Context decoupled
 - Overhead: Context might pass data that Strategy doesn't need
- Second option:
 - Strategy keeps a reference to the Context and requests data explicitly
 - More complex interface for Context
 - Strategy and Context are more coupled

Behavioral patterns: Visitor

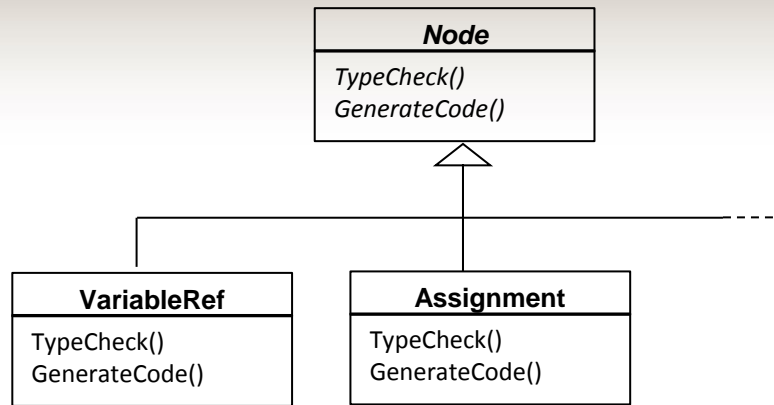
Visitor: motivation

Claudia Seardine

- Suppose you have to perform the same operation on different classes of an object structure
- Example:
 - A compiler takes as input a text file containing the program
 - Then it parses the file and creates as an *abstract syntax tree*
 - The tree contains nodes like variables, assignments, etc.
 - Finally, it performs operations on each node
 - Operations are static semantic analysis, code generation, etc.
- Complex structure consisting of different types of nodes and distinct operations

Visitor: motivation

- Naïve approach: a method for each operation



Problems:

- The system is hard to understand, maintain and change
- It doesn't scale (to add an operation we must change all nodes)
- Too many things in each single class

Solution:

- decouple the nodes from visiting
- encapsulate the various visiting operations

Visitor pattern

Claudia Scardine

Represent an operation to be performed on the elements of an object structure.

Visitor lets you define a new operation without changing the classes of the elements on which it operates.

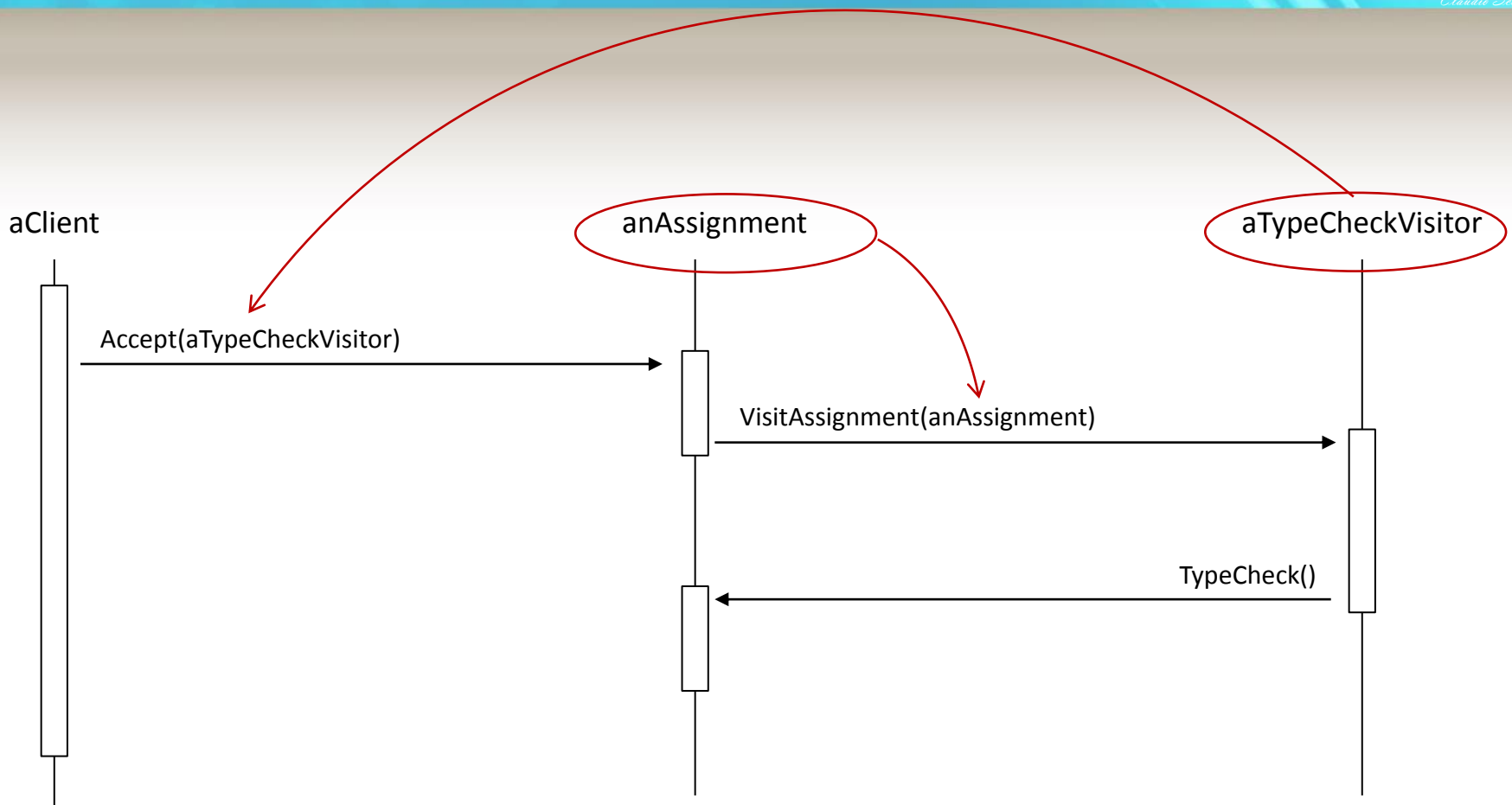
Visitor: example

- Idea:
 - We package related operations from each class in a separate object called **Visitor**
 - We pass the Visitor as argument to the abstract syntax tree elements
 - When an element accepts the visitor, it sends a request to the visitor including itself as argument
 - The Visitor will then execute the operation for that element



Visitor: example

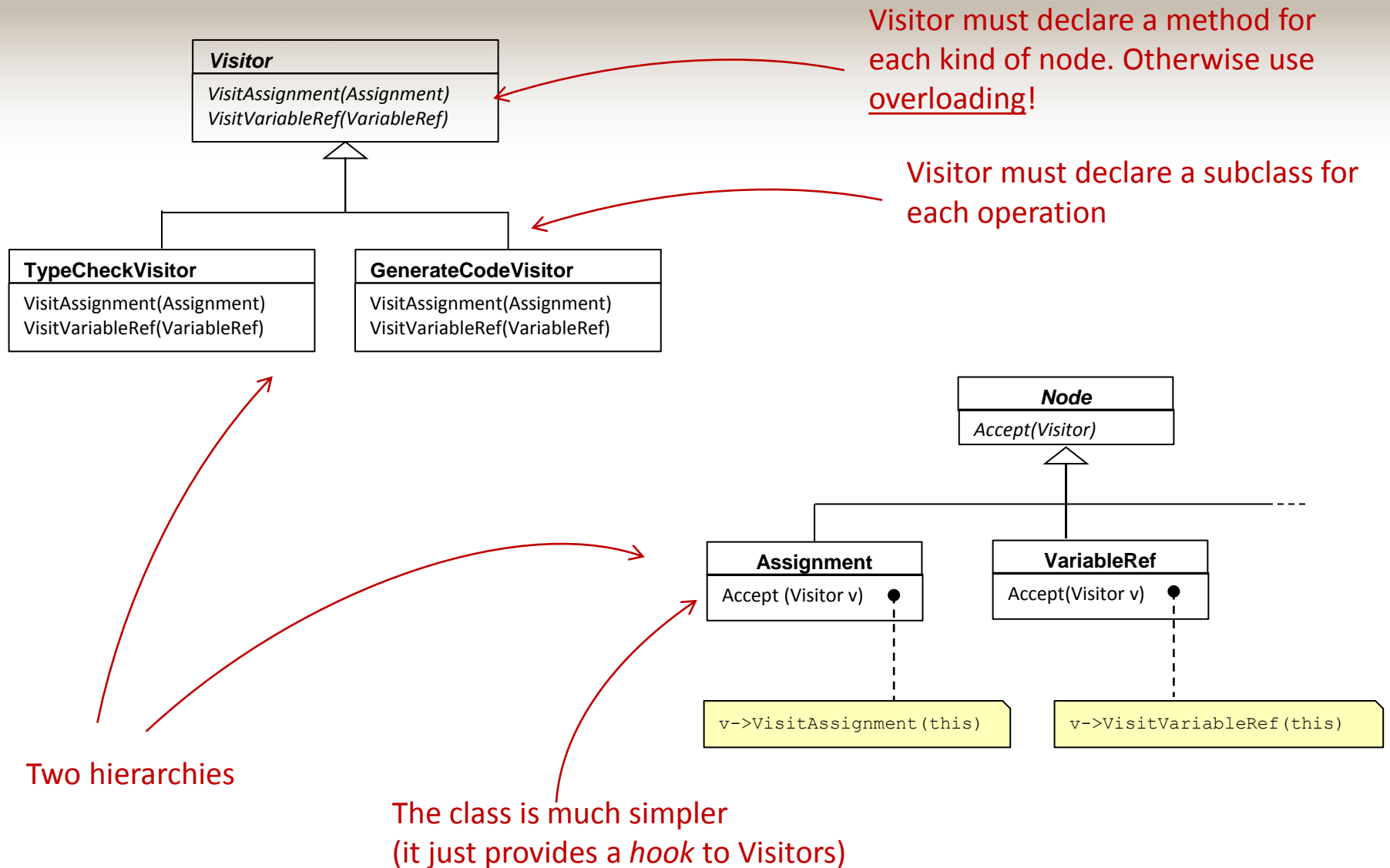
Claudia Scardine



“Double dispatch”: the operation that gets executed depends on both the type of Visitor (i.e., `aTypeCheckVisitor`) and the type of Element it visits (`anAssignment`). This is the key to the Visitor pattern.

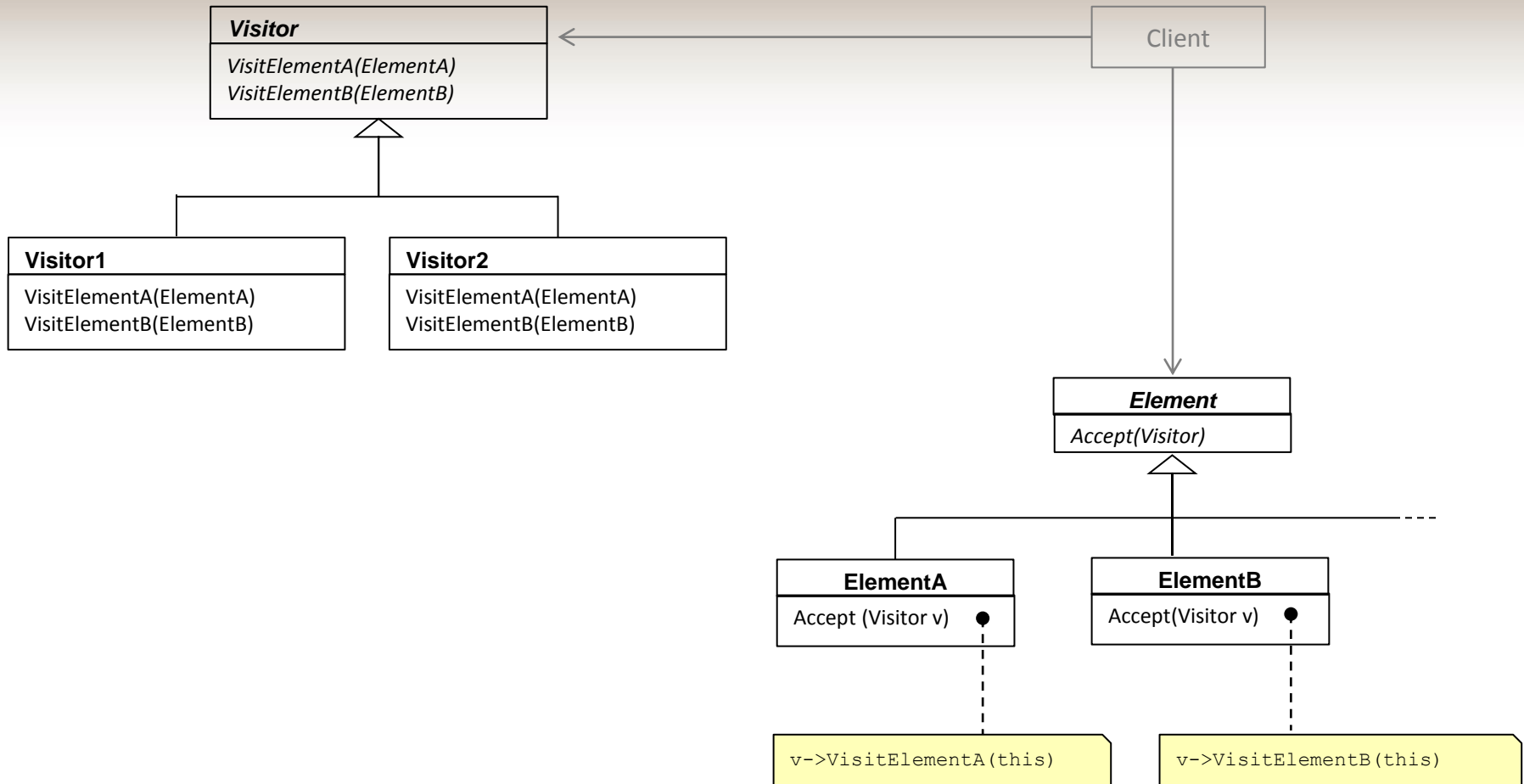
Visitor: example

Claudia Seardine



Visitor: diagram

Claudia Scardine



Visitor: applicability

Claudia Seardine

Use Visitor when:

- An object structure contains many classes of objects and you want to perform operations on these objects that depend on their concrete classes
- Many distinct and unrelated operations need to be performed on objects and you want to avoid “polluting” the classes with these operations.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure.

Visitor: participants

Claudia Seardine

Visitor:

- Declares a method for each class of elements in the object structure

Visitor1, Visitor2:

- Implement each method declared by Visitor
- Each method implements a fragment of the algorithm defined for the corresponding class
- It often accumulates results during the traversal of the structure

Element:

- Declares an Accept method that takes a Visitor as an argument

ElementA, ElementB:

- Implements the Accept method

Visitor: pros & cons

Claudia Seardine

Pros:

- Adding new operations is easy (just add a new visitor)
- A visitor gathers related operations and separates unrelated ones
- Visitor can visit objects that don't have a common parent (unlike the Iterator pattern)
- Visitors can accumulate a state inside the class instead of using global variables

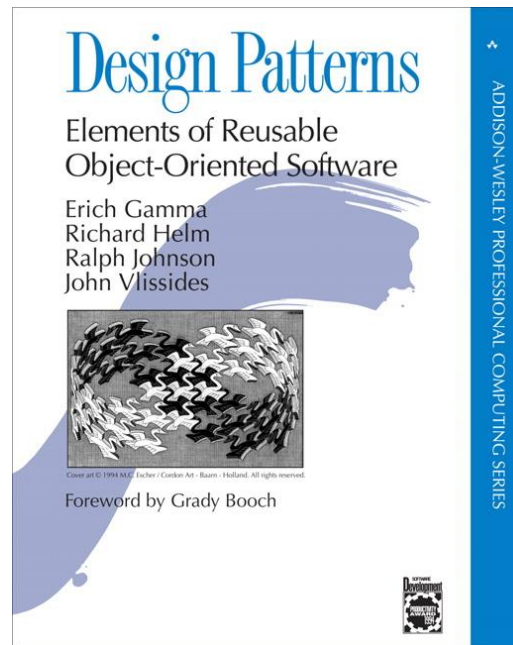
Cons:

- Adding new Element subclasses is hard
- Breaking encapsulation: the pattern often forces you to provide public methods to access element's internal state

Final remark...

Claudia Seardine

There are other 12 patterns in the book!



Questions ?

Claudia Scardine

