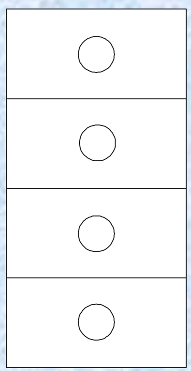
Outline

- Adaptor pattern
- Bridge pattern

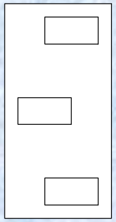
Adaptor

- Motivation
  An adaptor: device that connects pieces of equipment that were not originally designed to be connected

Application required interface



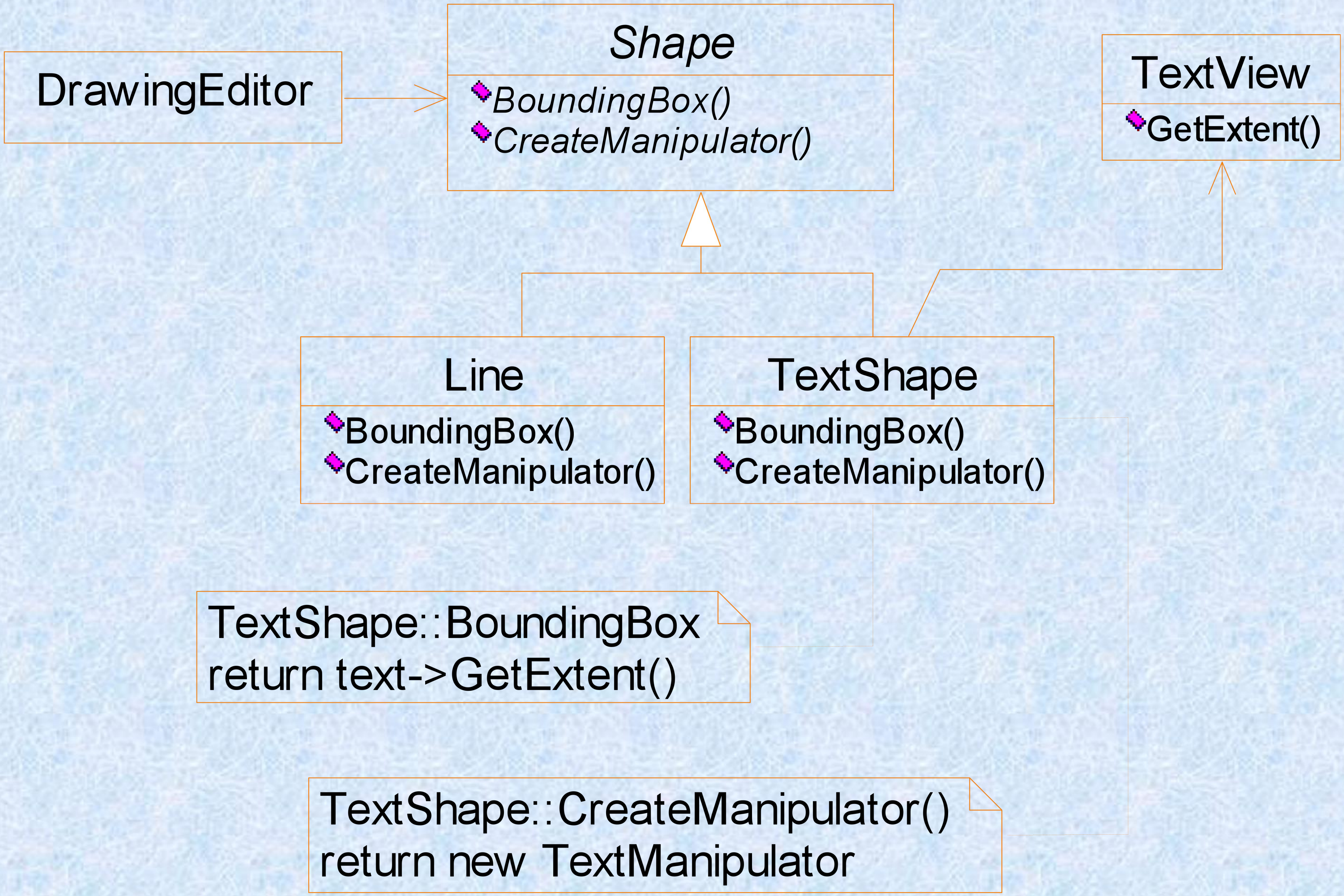
Adaptor



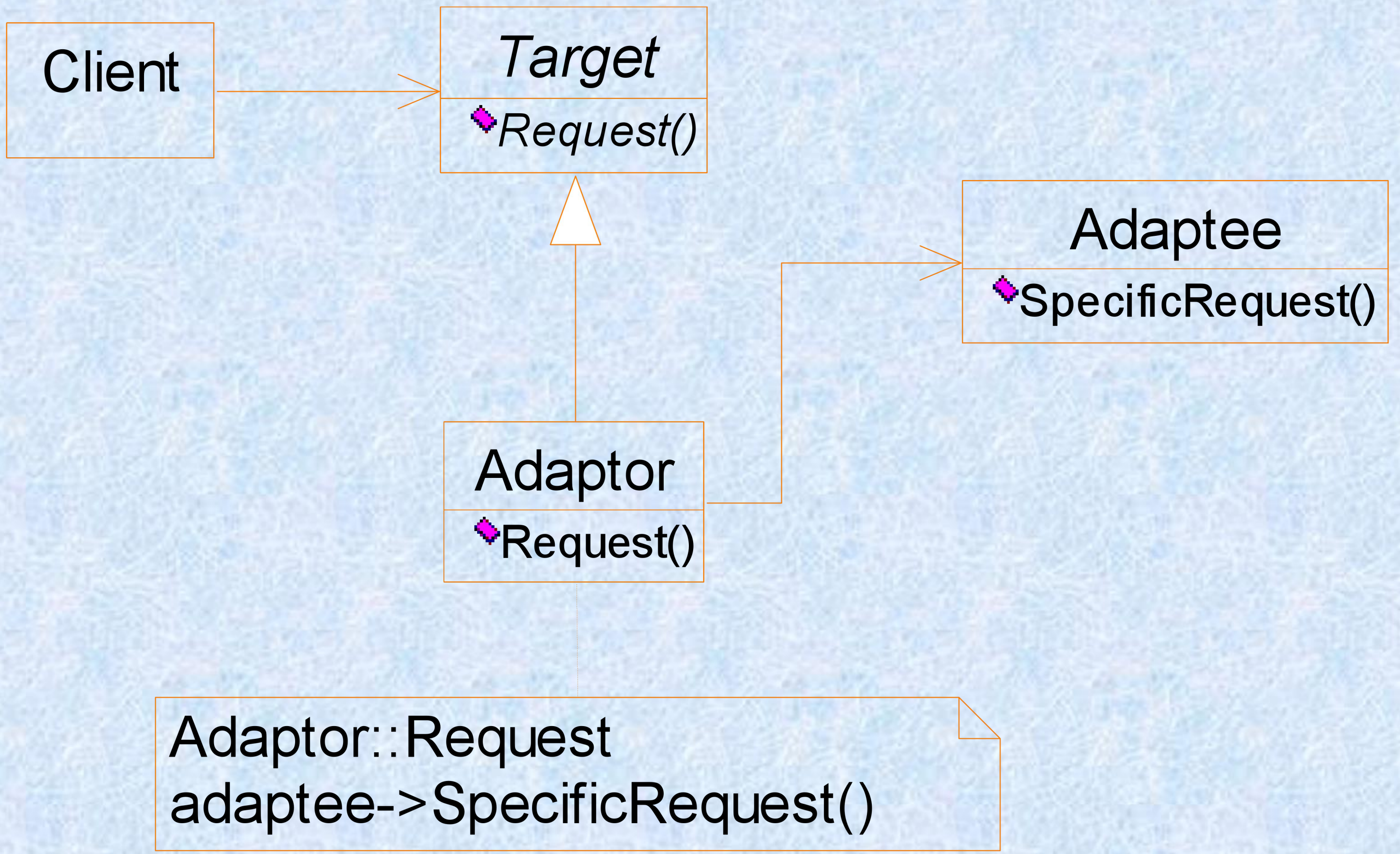
Toolkit class

An example

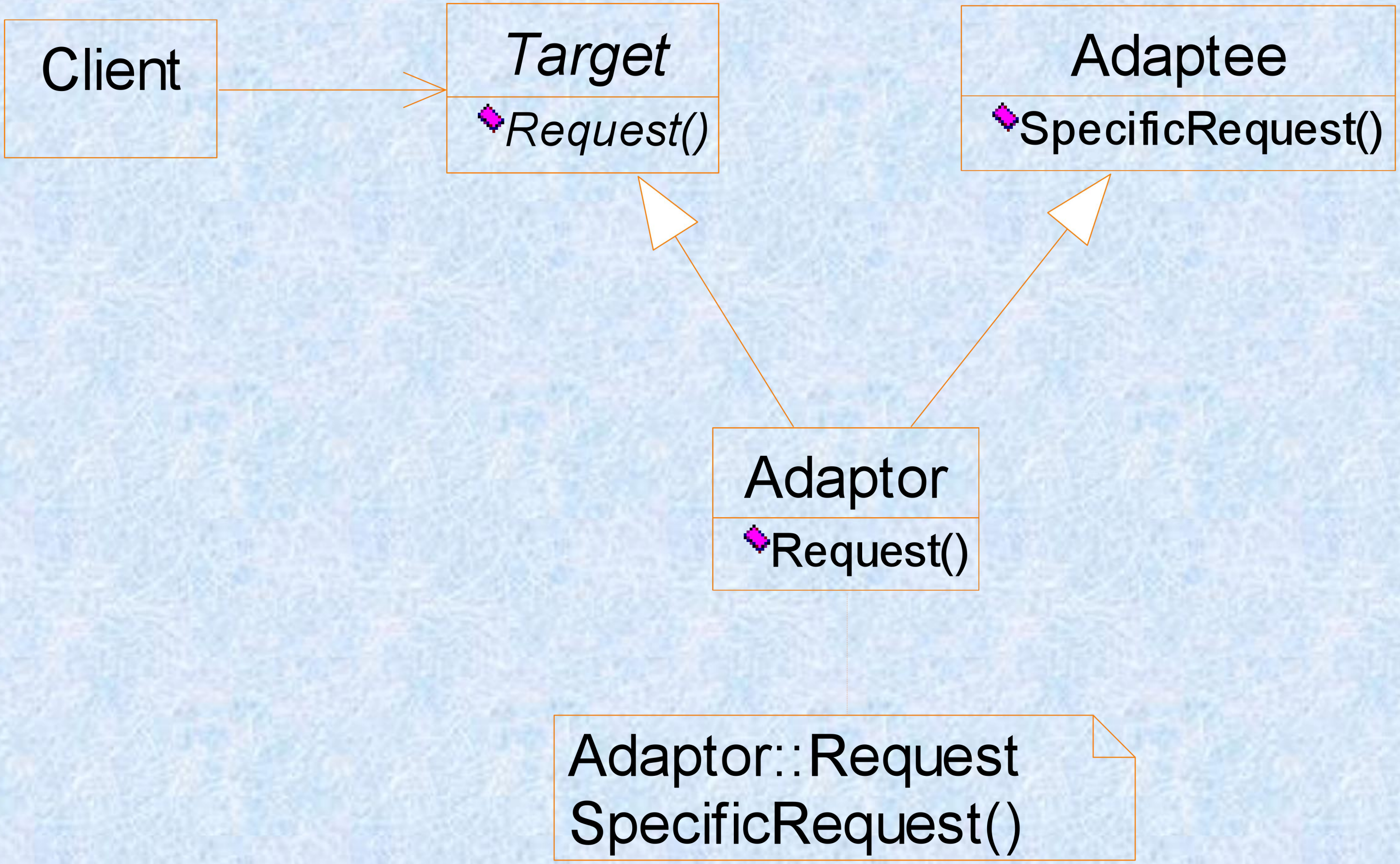
- Application: a drawing editor
- We have an abstract class Shape
- Geometric shapes are easy to implement
- Text shape is difficult to implement
  - displaying
  - Editing
- We want to reuse existing TextView class
- We can not or do not wish to change TextView's interface



An example of using the adaptor pattern



General structure: by class composition



General structure: by multiple class inheritance

Sample code

- For the multiple inheritance scheme
  - Inherit the interface publicly
  - Inherit the implementation privately

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(Point & bottomLeft,
                             Point & topRight ) const;
    virtual Manipulator * CreateManipulator() contst;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord&x, Coord&y) const;
    void GetExtent(Corrd &width, Coord&height) const;
    virtual bool IsEmpty() const;
}
```

```
class TextShape: public Shape, private TextView {
public:
    TextShape();
    virtual void BoundingBox( Point & bottomLeft,
        Point & topRight ) const{
        Corrd bottom, left, width, height;

        GetOrigin(bottom, left);
        GetExtent(width, height);

        bottomLeft = Point(bottom, left);
        topRight = Point(bottom + height, left + width);
    }
    // to be continued
```

```
 virtual bool IsEmpty() const {
        return TextView::IsEmpty();
    }
 virtual Manipulator* CreateManipulator() const {
        return new TextManipulator(this);
    }
}
```

Consequence

- The adaptor should append any necessary functionality that the adaptee does not have.
- The amount of work Adaptor does depends on how similar the Target interface is to Adaptee's.
- Difference between the two schemes
  - Class composition: a single adaptor works with many adaptees: the Adaptee itself and all its subclasses
  - Multiple inheritance: don't have the property

Implementation

- Pluggable adaptors
  - Be able to adapt to adaptees with different interfaces.

关于lambda表达式

一个lambda表达式本质上是一个匿名函子。C++11允许在STL算法中使用lambda表达式。例如

```
vector<string> lines;
sort(lines.begin(), lines.end(),
    [ ](string s1, string s2) -> bool {
            return s1.size() > s2.size( );
     }
);
```

Lambda表达式的语法

[capture](parameters) -> return_type { function_body }

capture部分表示将要以何种方式访问哪些外部对象。常见形式：

[ ] // 不访问任何外部对象

[&] // 函数体中对外部对象的访问默认为引用方式

[=] // 函数体中对外部对象的访问默认为值传递方式

[x, &y] // 以值传递的方式访问外部x，以引用方式访问外部y

[&, x] // 以值传递的方式访问外部x，以引用方式访问其它外部对象

[=, &z] // 默认为值传递方式，但以引用方式访问z

[this] // 甚至可以访问类的私有数据成员

可以捕获外部的局部对象。

如果参数为空，“()”也可以省略。不过不建议这么做。

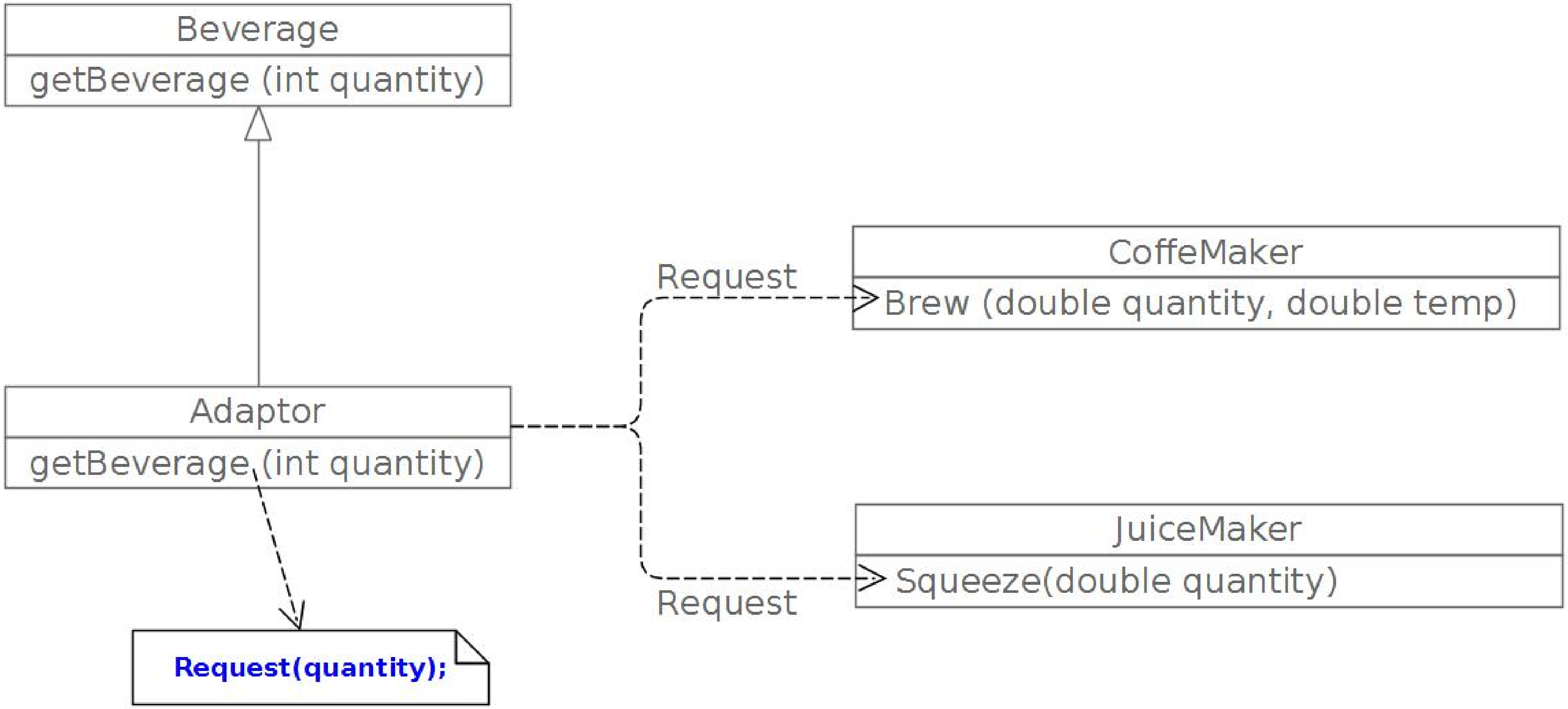
如果函数体中所有return语句的返回类型是一致的，return_type部分可以省略。

closure。闭包。指运行阶段一个lambda表达式对应的一个对象。

std::function

是一个类模版，表示一个函子。

```
void print_num(int i){...}
struct PrintNum {
    void operator()(int i) const {
        std::cout << i << '\n';
    }
};
int main() {
      std::function<void(int)> f1 = print_num;
      std::function<void()>     f2 = [ ]( ) { print_num(42); };
      std::function<void(int)> f3 = PrintNum();
}
```


Beverage
getBeverage (int quantity)
CoffeMaker
Request
Brew (double quantity, double temp)
Adaptor
getBeverage (int quantity)
JuiceMaker
Squeeze(double quantity)
Request
Request(quantity);

```
#include <iostream>
#include <functional>
//Adaptee 1
struct CoffeMaker {
    void Brew (double quantity, double temp) const {
            std::cout << "I am brewing " << quantity <<"ml coffee @" << temp
                        <<" degree C" <<std::endl;
    }
};
//Adaptee 2 (having difference interface from Adaptee 2)
struct  JuiceMaker{
    void Squeeze (double quantity) const {
            std::cout << "I am making " << quantity <<"ml Juice" <<std::endl;
    }
};
// Target
struct Beverage{
     virtual void getBeverage (int quantity) = 0;
};
```

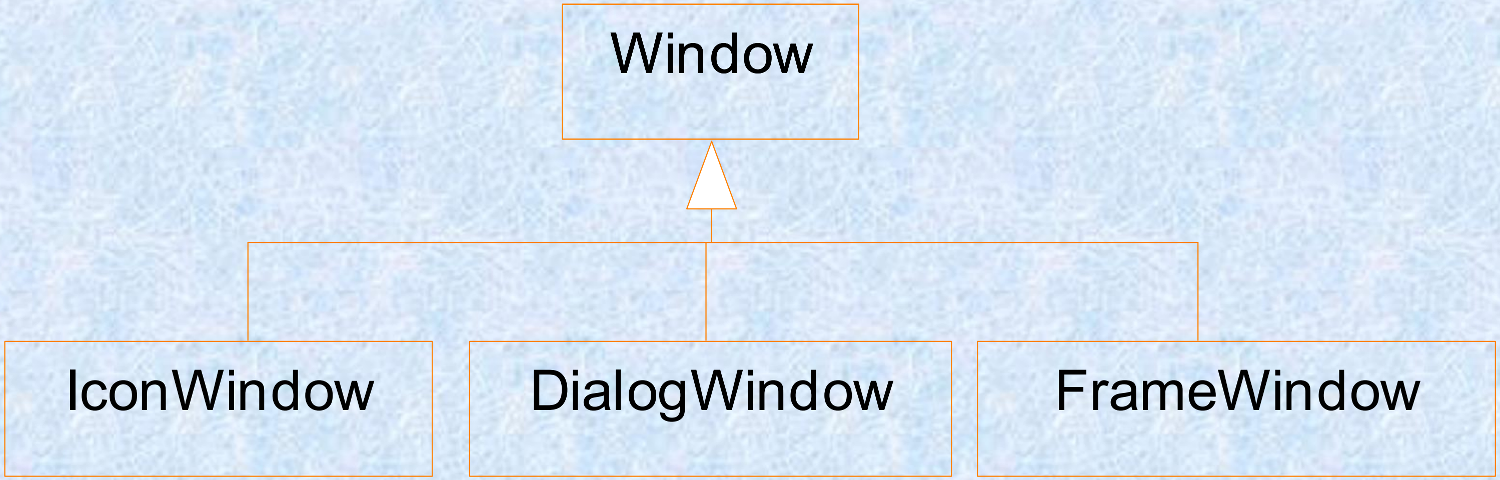
```
// Adapter
class Adapter : public Beverage  {
     std::function<void(int)> Request;
public:
    Adapter(CoffeMaker *cm1){
         Request = [cm1](int quantity) {
                   cm1->Brew(quantity,80);
          };
    }

    Adapter(JuiceMaker *jm1){
          Request = [jm1](int quantity){
                 jm1->Squeeze(quantity);
          };
     }

     void getBeverage(int quantity){
              Request(quantity);
     }
};
```

```
//Client
int main() {
    CoffeMaker   coffeMaker;
    Adapter adp1 ( & coffeMaker()  );
    adp1.getBeverage(30);

    JuiceMaker   juiceMaker
    Adapter adp2 ( & juiceMaker  );
    adp2.getBeverage(40);

}
```

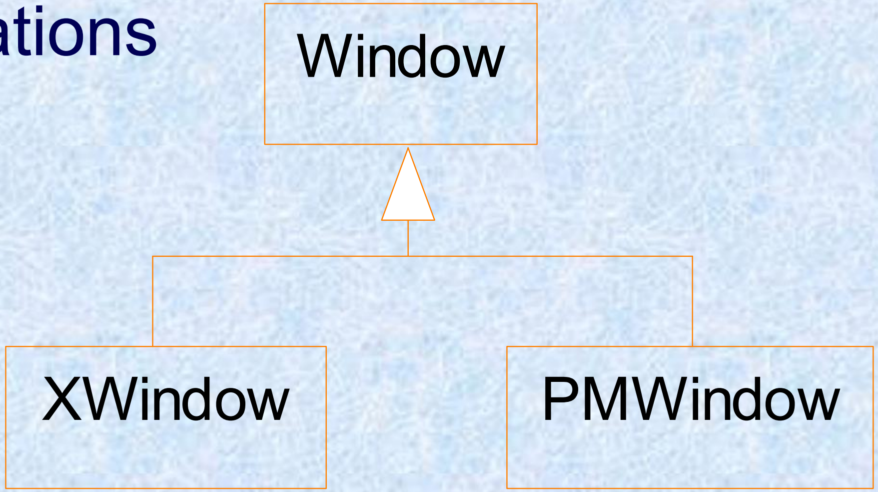
Bridge Pattern

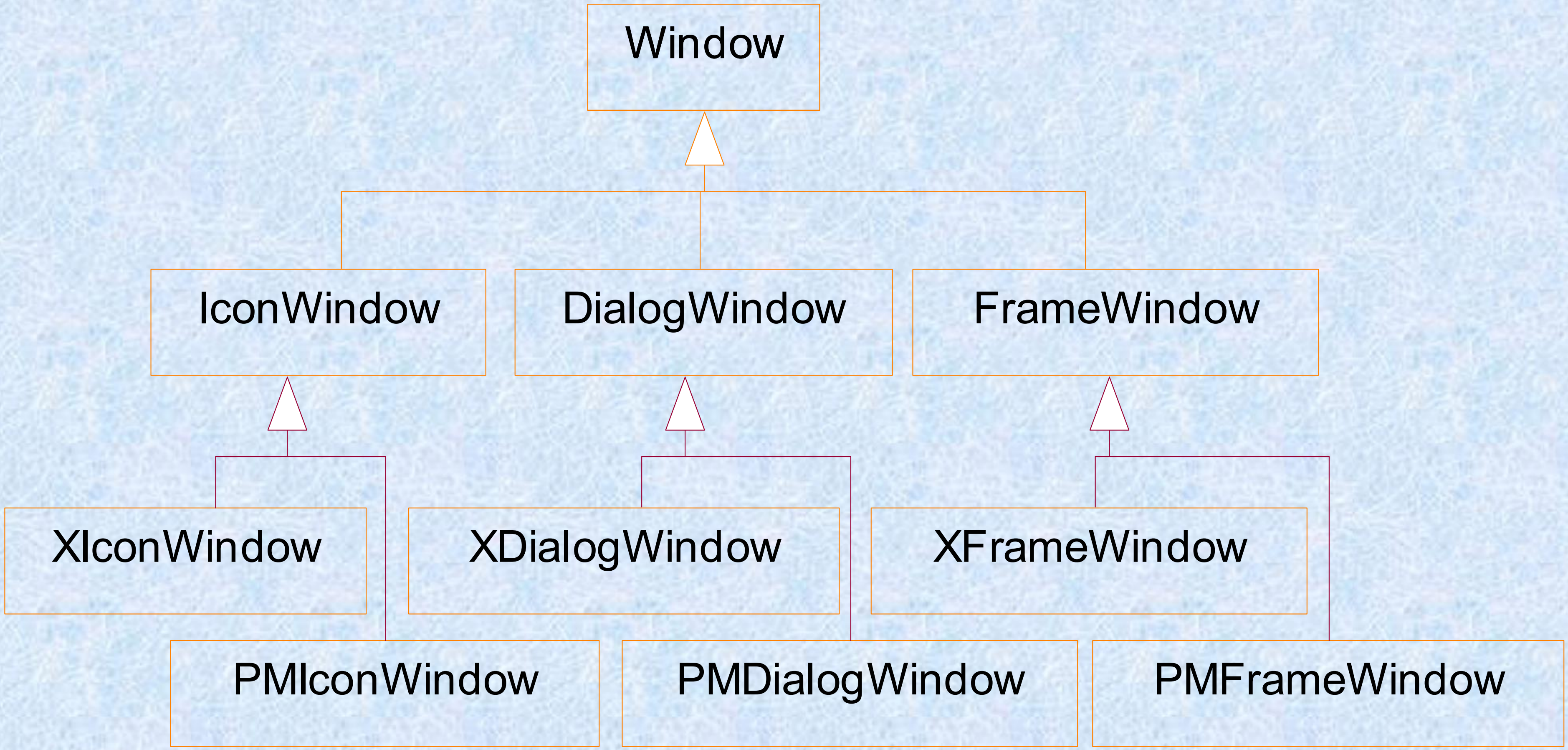
- Motivation
  - When an abstraction have several possible implementation
  - Usually we use inheritance
- An example
  - A portable Window for X and PM
  - Drawbacks of using inheritance
    - Two many classes
    - Make client code platform-dependent
      We may use abstract factory pattern here.

Different Kinds



Different Implementations

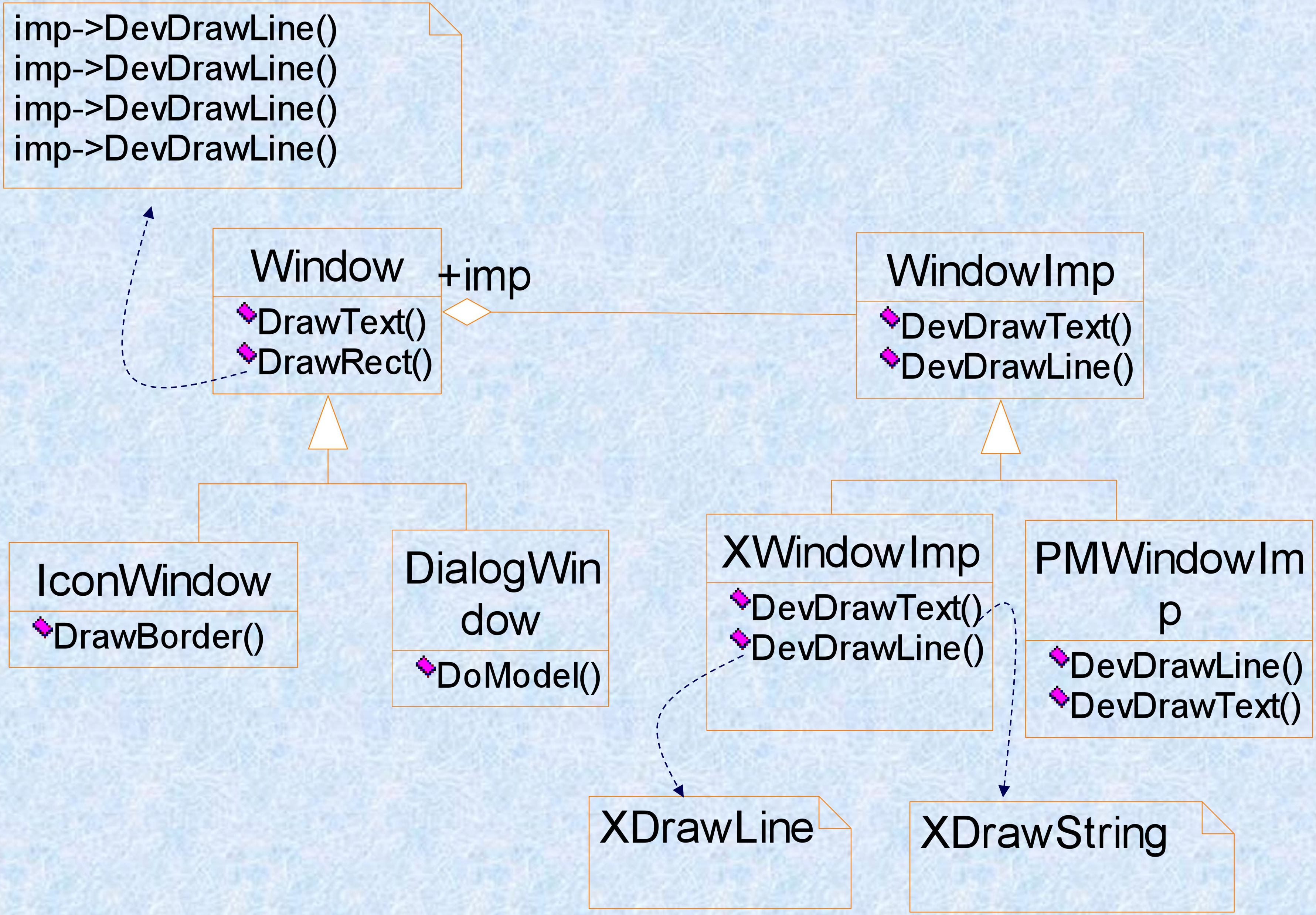




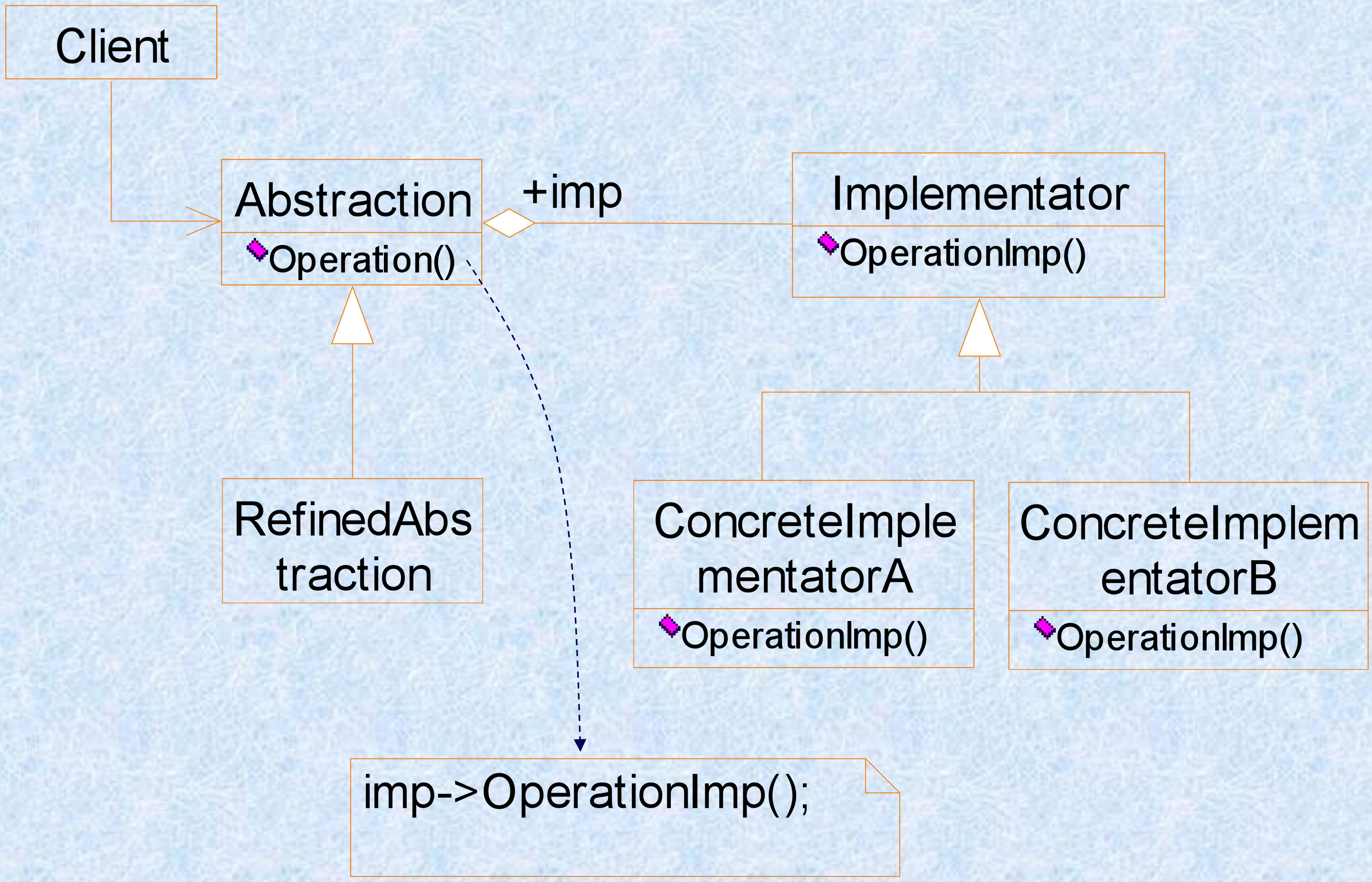
To cover all kinds of windows and all platforms

The bridge pattern

- There are two factors: window kinds and window implementation
- We use two separate class hierarchies
  - One to model different kinds of windows
  - Another to model the detailed implementation of the windows
    We need to find a “narrow” interface for all possible operations.



Using the bridge pattern



General structure of the bridge pattern

Discussion on bridge pattern

- Both the abstraction hierarchy and the implementation hierarchy can be extended by subclassing
- Changes in the implementation does not affect the abstract (no recompilation required)
- Difference between bridge and strategy patterns
  - Strategy pattern: run-time switch, for an operation, has context → a behavioural pattern
  - Bridge pattern: class hierarchy → a structural pattern