Laboratorio di Tecnologie dell'Informazione

Ing. Marco Bertini
bertini@dsi.unifi.it
http://www.dsi.unifi.it/~bertini/

Design patterns

What are design patterns ?

- In software engineering, a design pattern is a general reusable solution to a commonly occurring problem in software design.
  - A design pattern is not a finished design that can be transformed directly into code.
  - It is a description or template for how to solve a problem that can be used in many different situations.
- Patterns are the recurring solutions to the problems of design.

When DPs were developed ?

- The idea originates from a book that organized implicit knowledge about how people solve recurring problems in building things:

"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" - Prof. Charles Alexander

Why using design patterns ?

- Design patterns can speed up the development process by providing tested, proven development paradigms.
- Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.
- Design patterns allow developers to communicate using well-known, well understood names for software interactions.

"Gang of Four" Pattern Structure

- Gang of Four (GoF): Gamma, Johnson, Helm, Vlissides
- Authors of the popular "Design Patterns" book
- A pattern has a **name**
  - e.g., the Command pattern
- A pattern documents a recurring **problem**
  - e.g., issuing requests to objects without knowing in advance what's to be requested or of what object
- A pattern describes the core of a **solution**
  - e.g., class roles, relationships, and interactions
  - Important: this is different than describing a design
- A pattern considers **consequences** of its use
  - Trade-offs, unresolved forces, other patterns to use

What patterns are not

- Design patterns ...
  - ... are not restricted to OOP
  - ... are not heuristics or abstract principles
  - ... are not new and/or untested solutions
  - ... are not a specific solution to a specific problem
  - ... are not a “silver bullet”

The iterator design pattern

- We have already seen an example of behavioural design pattern: the iterators of STL.
  - The basic idea of the iterator is that it permits the traversal of a container (like a pointer moving across an array). However, to get to the next element of a container, you need not know anything about how the container is constructed.
    This is the iterators job. By simply using the member functions provided by the iterator, you can move, in the intended order of the container, from the first element to the last element.

Design patterns classification

- Design patterns were originally grouped into the categories Creational patterns, Structural patterns, and Behavioral patterns.
- DPs were described using the concepts of:
- **delegation:** is the concept of handing a task over to another part of the program. In object-oriented programming it is used to describe the situation wherein one object defers a task to another object, known as the delegate.
- **aggregation:** is a way to combine simple objects or data types into more complex ones. Composited objects are often referred to as having a "has a" relationship. In composition, when the owning object is destroyed, so are the contained objects. In aggregation, this is not necessarily true.
- **consultation:** is when an object's method implementation consists of a message send of the same message to another constituent object.

Types of design patterns - I

Creational design patterns

- This design patterns is all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns.
- While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Types of design patterns - 2

Structural design patterns

- This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces.
- Structural object-patterns define ways to compose objects to obtain new functionality.

Types of design patterns - 3

Behavioral design patterns

- This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

Some patterns

- Observer - behavioral - A way of notifying change to a number of classes
- Factory - creational - Creates an instance of several families of classes
- Singleton - creational - A class of which only a single instance can exist
- Adapter - structural - Match interfaces of different classes
- Facade - structural - A single class that represents an entire subsystem
- Decorator - structural - Add responsibilities to objects dynamically

Programming idioms

- Idioms are reoccurring solutions to common programming problems.
- Idioms are low-level patterns specific to a programming language. Design patterns are high-level and language independent.
- During implementation you look for idioms. During design you look for patterns.

We have already seen them...

- include guard, RAII, const auto_ptr, are programming idioms
- Another example: the interface class
- if we are more interested in interface inheritance than implementation inheritance we design the interface using a class composed only by pure virtual public methods

Idiom: interface class

```
   class Shape   // An interface class
{
  public:
    virtual ~Shape();
    virtual void move_x(int x) = 0;
    virtual void move_y(int y) = 0;
    virtual void draw() = 0;
//...
};

class Line : public Shape
{
  public:
    virtual ~Line();
    virtual void move_x(int x); // implements move_x
    virtual void move_y(int y); // implements move_y
    virtual void draw(); // implements draw
  private:
    point end_point_1, end_point_2;
//...
};
```

Idiom: interface class

```
   class Shape   // An interface class
{
  public:
    virtual ~Shape();
    virtual void move_x(int x) = 0;
    virtual void move_y(int y) = 0;
    virtual void draw() = 0;
//...
};

class Line : public Shape
{
  public:
    virtual ~Line();
    virtual void move_x(int x); // implements move_x
    virtual void move_y(int y); // implements move_y
    virtual void draw(); // implements draw
  private:
    point end_point_1, end_point_2;
//...
};
```

In Java we would use an `Interface`

Idiom: interface class

```
   class Shape   // An interface class
{
  public:
    virtual ~Shape();
    virtual void move_x(int x) = 0;
    virtual void move_y(int y) = 0;
    virtual void draw() = 0;
//...
};

class Line : public Shape
{
  public:
    virtual ~Line();
    virtual void move_x(int x); // implements move_x
    virtual void move_y(int y); // implements move_y
    virtual void draw(); // implements draw
  private:
    point end_point_1, end_point_2;
//...
};
```

Idiom: interface class

```
class Shape   // An interface class
{
  public:
    virtual ~Shape();
    virtual void move_x(int x) = 0;
    virtual void move_y(int y) = 0;
    virtual void draw() = 0;
//...
};

class Line : public Shape
{
  public:
    virtual ~Line();
    virtual void move_x(int x); // implements move_x
    virtual void move_y(int y); // implements move_y
    virtual void draw(); // implements draw
  private:
    point end_point_1, end_point_2;
//...
};
```

Remind the virtual destructor !

Idiom: interface class

```
class Shape   // An interface class
{
  public:
    virtual ~Shape();
    virtual void move_x(int x) = 0;
    virtual void move_y(int y) = 0;
    virtual void draw() = 0;
//...
};

class Line : public Shape
{
  public:
    virtual ~Line();
    virtual void move_x(int x); // implements move_x
    virtual void move_y(int y); // implements move_y
    virtual void draw(); // implements draw
  private:
    point end_point_1, end_point_2;
//...
};
```

Remind the virtual destructor !

The classes that extend Shape are not dependent each other

Adapter pattern

Class and Object Adapter

Class and Object Adapter

- Review the pattern we have already seen in the slides on inheritance: the class adapter and see another version of the pattern, the object adapter
- The Adapter pattern converts the interface of a class into another interface the clients expect.
  Adapter lets classes work together that could not otherwise because of incompatible interfaces.

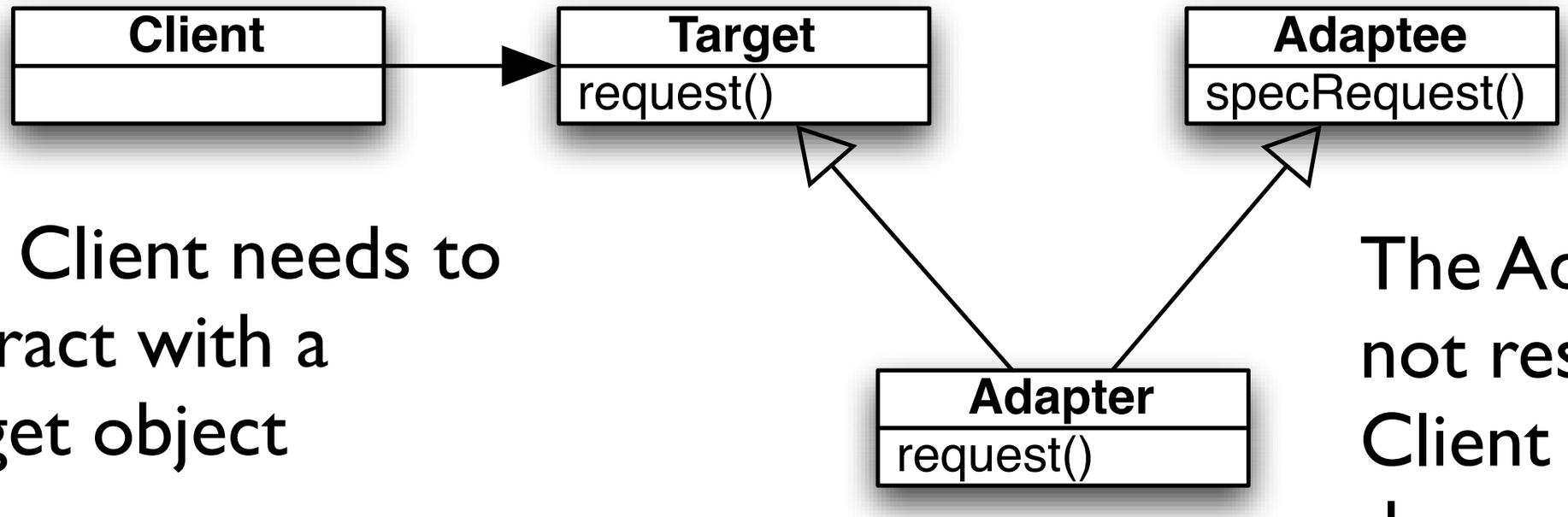
Adapter pattern

- Problem
  - Have an object with an interface that's close to but not exactly what we need
- Context
  - Want to re-use an existing class
  - Can't change its interface
  - Impractical to extend class hierarchy more generally
- Solution
  - Wrap a particular class or object with the interface needed (2 forms: class form and object forms)
- Consequences
  - Implementation you're given gets interface you want

Class Adapter

- A class (Adapter) adapts the interface of another class (Adaptee) to a client, using the expected interface described in an abstract class (Target)
- Uses multiple inheritance, along with abstract class, virtual methods and private inheritance.

Class Adapter UML class



The Client needs to interact with a Target object

The Adaptee could not respond to Client because it does not have the required method

The Adapter lets the Adaptee to respond to request of a Target by extending both Target and Adaptee

Class Adapter example

```
class Adaptee {
public:
  getAlpha() {return alpha;};
  getRadius() {return radius;};
private:
  float alpha;
  float radius;
};

class Target {
public:
  virtual float getX() = 0;
  virtual float getY() = 0;
};
```

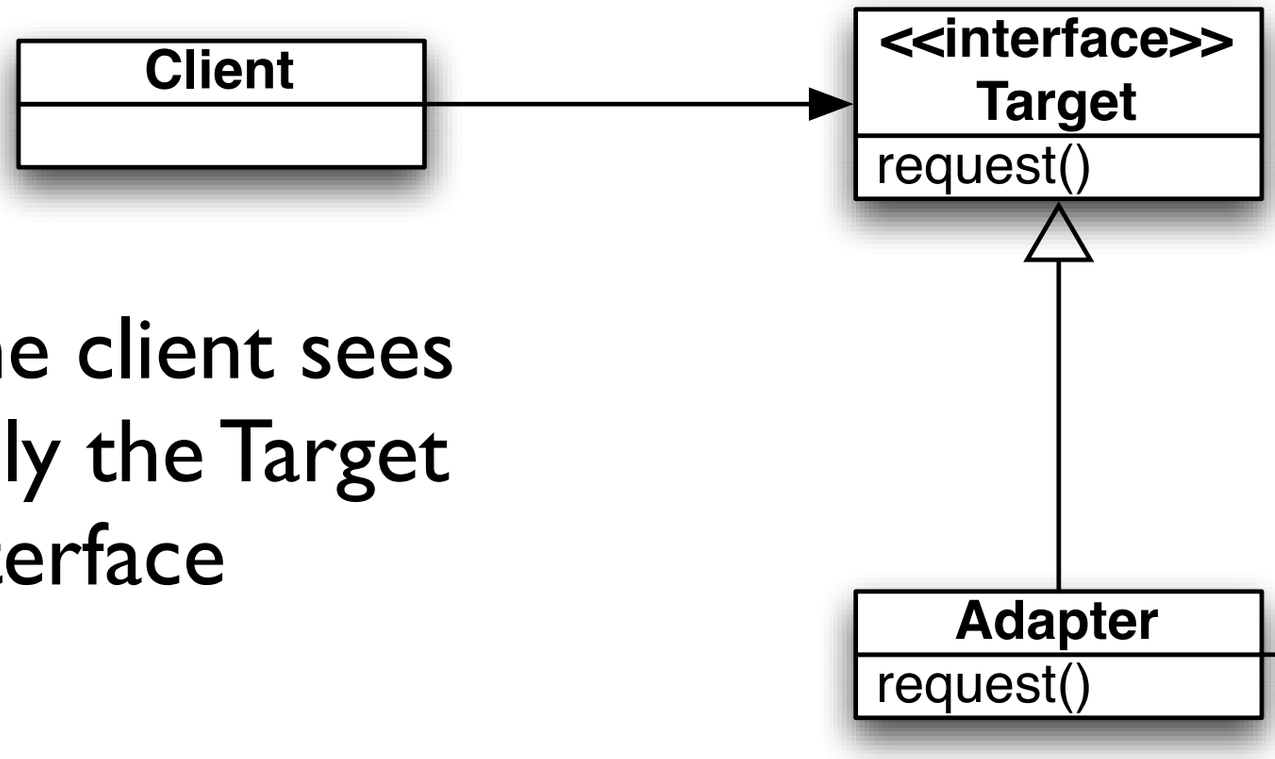
```
class Adapter : private Adaptee, public Target
{
public:
  virtual float getX();
  virtual float getY();
};
float Adapter::getX() {
 return
Adaptee::getRadius()*cos(Adaptee::getAlpha());
}
float Adapter::getY() {
 return
Adaptee::getRadius()*sin(Adaptee::getAlpha());
}
```

The Client can't access Adaptee methods because Adapter has obtained them using private inheritance

Object Adapter

- Does not use multiple inheritance but uses composition to adapt the adaptee class:
  - it can adapt also subclasses of the adaptee (unlike the Class Adapter)
  - requires to reimplement the adaptee, thus require more coding (unlike the Class Adapter)
- The adapter delegates to the adaptee the requests of the clients

Object Adapter UML class diagram



The client sees only the Target interface

The Adapter implements the Target interface

The Adapter is composed with the Adaptee

The Adapter delegates all the requests to the Adaptee

Object Adapter example

```
class Duck {
  public: virtual ~Duck() = 0 {   }
  public: virtual void fly() const = 0;
  public: virtual void quack() const = 0;
};

class MallardDuck : public Duck {
  public: void fly() const {
    cout << "I'm flying" << std::endl;
  }
  public: void quack() const {
    cout << "Quack" << std::endl;
  }
};

class Turkey {
  public: virtual ~Turkey() = 0 { }
  public: virtual void gobble() const = 0;
  public: virtual void fly() const = 0;
};
```

```
class WildTurkey : public Turkey {
  public: void fly() const {
    cout << "I'm flying a short
distance" << endl;
  }
  public: void gobble() const {
    cout << "Gobble gobble" << endl;
  }
};
```

Object Adapter example

```
class TurkeyAdapter : public Duck {
  private: const Turkey* _turkey;
  public: TurkeyAdapter( const Turkey* turkey ) : _turkey( turkey ) { }
  public: void fly() const {
    for( int i = 0; i < 5; i++ ) {
      _turkey->fly();
    }
  }
  public: void quack() const {
    _turkey->gobble();
  }
};
```

```
  public: void quack() const {
    cout << "Quack" << std::endl;
  }
};

class Turkey {
  public: virtual ~Turkey() = 0 { }
  public: virtual void gobble() const = 0;
  public: virtual void fly() const = 0;
};
```

```
    cout << "Gobble gobble" << endl;
  }
};
```

Object Adapter example

```
class TurkeyAdapter : public Duck {
  private: const Turkey* _turkey;
  public: TurkeyAdapter( const Turkey* turkey ) : _turkey( turkey ) { }
  public: void fly() const {
    for( int i = 0; i < 5; i++ ) {
      _turkey->fly();
    }
  }
  public: void quack() const {
    _turkey->gobble();
  }
};
```

Here's the composition ! Any class extending Turkey may be adapted because of the IS_A relationship

```
  public: void quack() const {
    cout << "Quack" << std::endl;
  }
};

class Turkey {
  public: virtual ~Turkey() = 0 { }
  public: virtual void gobble() const = 0;
  public: virtual void fly() const = 0;
};
```

Object Adapter example

```
class TurkeyAdapter : public Duck {
  private: const Turkey* _turkey;
  public: TurkeyAdapter( const Turkey* turkey ) : _turkey( turkey ) { }
  public: void fly() const {
    for( int i = 0; i < 5; i++ ) {
      _turkey->fly();
    }
  }
  public: void quack() const {
    _turkey->gobble();
    }
  };
```

Here's the delegation

Here's the composition ! Any class extending Turkey may be adapted because of the IS_A relationship

```
  public: void quack() const {
    cout << "Quack" << std::endl;
  }
};

class Turkey {
  public: virtual ~Turkey() = 0 { }
  public: virtual void gobble() const = 0;
  public: virtual void fly() const = 0;
};
```

Object Adapter example

```
class TurkeyAdapter : public Duck {
  private: const Turkey* _turkey;
  public: TurkeyAdapter( const Turkey* turkey ) : _turkey( turkey ) { }
  public: void fly() const {
    for( int i = 0; i < 5; i++ ) {
      _turkey->fly();
    }
  }
  public: void quack() const {
    _turkey->gobble();
  }
};
```

Here's the composition !
Any class extending Turkey may be adapted because of the IS_A relationship

Here's the delegation

cout

```
void testDuck( const Duck* duck ) {
     duck->quack();
     duck->fly();
}
...
MallardDuck* duck = new MallardDuck();
WildTurkey* turkey = new WildTurkey();
Duck* turkeyAdapter = new TurkeyAdapter( turkey );
testDuck( duck );
testDuck( turkeyAdapter );
...
```

Credits

- These slides are (heavily) based on the material of:
  - Aditya P. Mathur, Purdue University
  - Dr. Aaron Bloomfield, University of Virginia
  - Fred Kuhns, Washington University
  - Glenn Puchtel