Design Patterns

Overview

- What is a Design Pattern?
- Creational Patterns
- Structural Patterns
- Behavioral Patterns

References

- Erich Gamma et al., "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995
- Frank Buschmann et al., "Pattern-Oriented Software Architecture - A System of Patterns", Wiley, 1996
- Steven John Metsker, "Design Patterns Java™ Workbook", Addison-Wesley, 2002

What Is a Design Pattern?

Christopher Alexander says:

"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 the same thing twice."

Essential Elements

- A pattern has four essential elements:
  - The pattern name that we use to describe a design problem,
  - The problem that describes when to apply the pattern,
  - The solution that describes the elements that make up the design, and
  - The consequences that are the results and trade-offs of applying the pattern.

Design Patterns Are Not About Design

- Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is.
- Design patterns are not complex, domain-specific designs for an entire application or subsystem.
- Design patterns are descriptions of *communicating objects and classes that are customized to solve a general design problem in a particular context.*

Describing Design Patterns

- A common way to describe a design pattern is the use of the following template:
  - Pattern Name and Classification
  - Intent
  - Also Known As
  - Motivation (Problem, Context)
  - Applicability (Solution)
  - Structure (a detailed specification of structural aspects)
  - Participants, Collaborations (Dynamics)
  - Implementation
  - Example
  - Known Uses
  - Consequences
  - Known Uses

Creational Patterns

- Creational patterns abstract the instantiation process. They help to make a system independent of how its objects are created, composed, and represented.
  - Creational patterns for classes use inheritance to vary the class that is instantiated.
  - Creational patterns for objects delegate instantiation to another object.

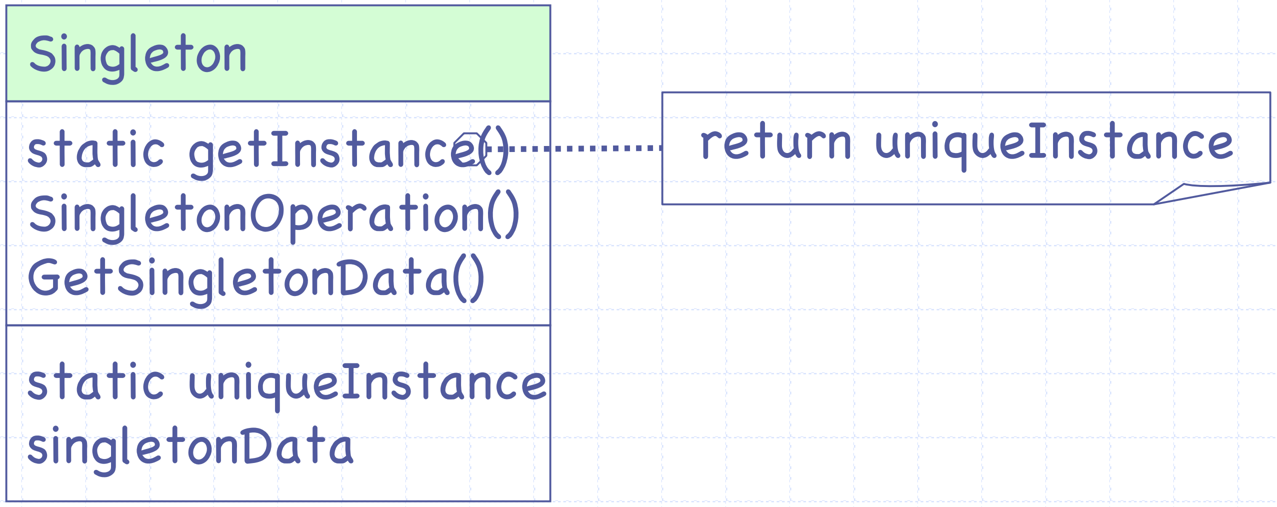
Why Creational Patterns?

- Creational patterns encapsulate knowledge about which concrete classes the system uses.
- Creational patterns hide how instances of these classes are created and put together.
- Creational patterns help to shift away from hard-coding a fixed set of behaviors towards defining a smaller set of fundamental behaviors that can be composed into any number of a more complex one.

Singleton

- Intent:
  - Ensure a class has only one instance, and provide a global point of access to it.
- Collaborations:
  - Clients access a singleton instance solely through singleton's instance operation.

Structure of Singleton



Implementation

```
class Singleton
{
private:
    // private static instance of a Singleton
    static Singleton* fInstance;

protected:
    Singleton() { … }            // protected constructor

    // public static getter to retrieve an instance of a singleton
public:
    static Singleton* getInstance()
    {
        if ( fInstance == null )
            fInstance = new Singleton();
        return fInstance;
    }
};
```

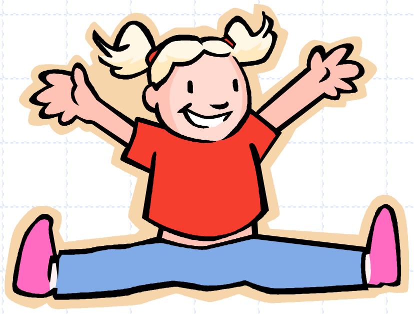
Default constructor!

Test Singleton

```
Singleton* aObj1 = Singleton.getInstance(); // Get first Singleton
Singleton* aObj2 = Singleton.getInstance(); // Get second Singleton

// test that aObj1 and aObj2 are indeed identical
if ( aObj1->equals( aObj2 ) )
    Console.WriteLine( "The objects are identical copies!" );
else
    Console.WriteLine( "OOPS! The objects are not identical copies!" );

// test that aObj1 and aObj2 are the same
if ( aObj1 == aObj2 )
    Console.WriteLine( "The objects are the same!" );
else
    Console.WriteLine( "OOPS! The objects are not the same!" );
```

Output

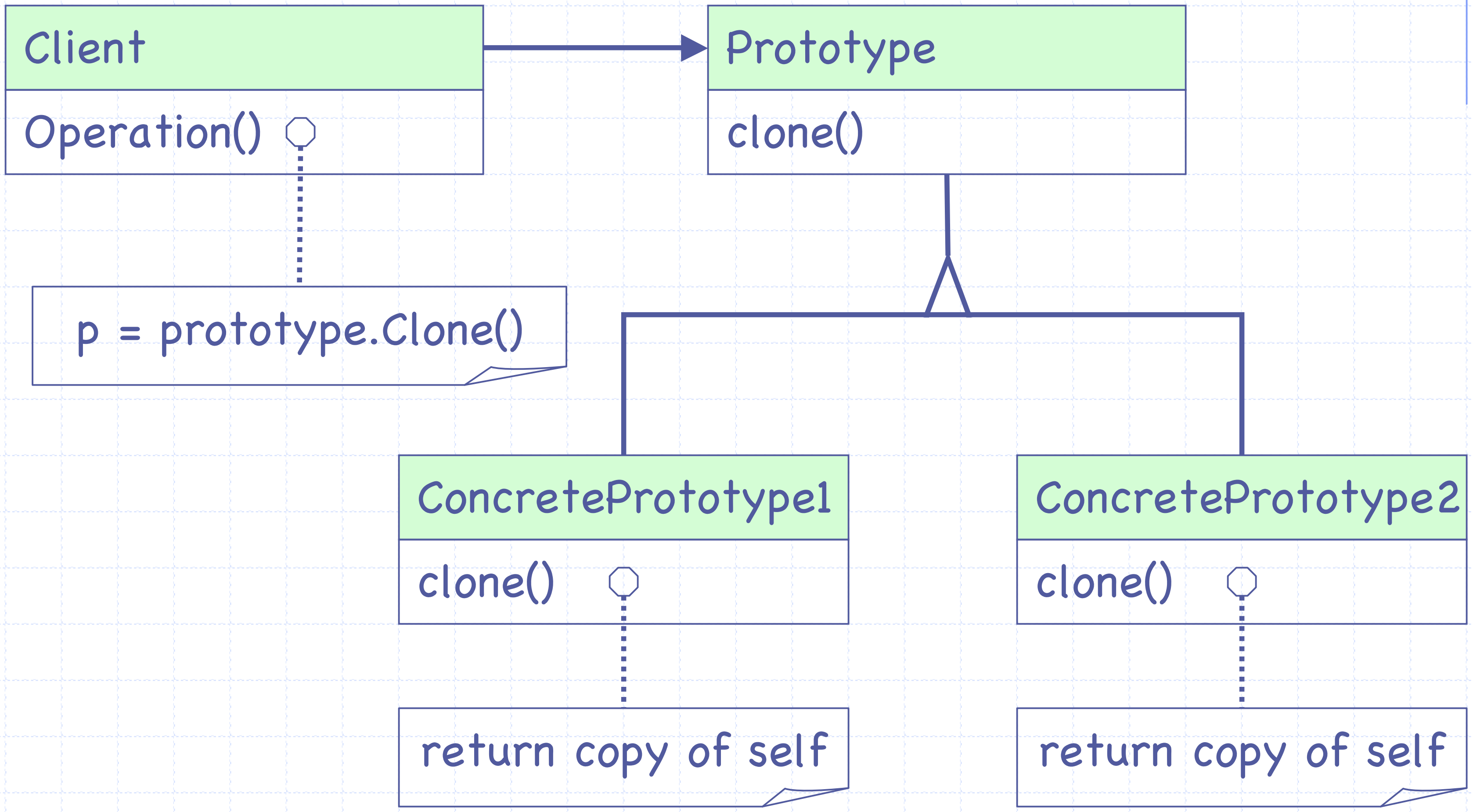
```
The objects are identical copies!
The objects are the same!
```



Prototype

- Intent:
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Collaborations:
  - A client asks a prototype to clone itself.

Structure of Prototype



Implementation

```
class Prototype
{
    friend SomeCreatorClass;

private:
    string fValue; // some instance variable

public:
    // Creates a new object that is a copy of the current instance.
    Prototype* clone() { return new Prototype( fValue ); }

protected:
    // Access is limited to the current class or derived types
    Prototype( string aValue ) { fValue = aValue; }
};
```

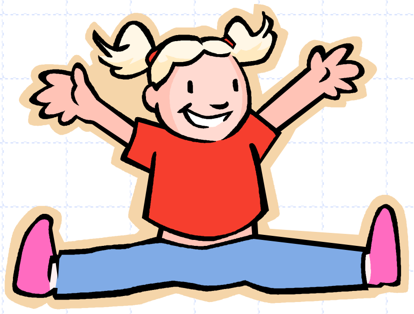
Test Prototype

```
Prototype* aObj1 = new Prototype( "A value" ); // Get first instance

// Get second instance using Prototype
Prototype* aObj2 = aObj1.clone();

// test that aObj1 and aObj2 are identical copies
if ( aObj1.equals( aObj2 ) )
    Console.WriteLine( "The objects are identical copies!" );
else
    Console.WriteLine( "OOPS! The objects are not identical copies!" );

// test that aObj1 and aObj2 are not identical
if ( aObj1 == aObj2 )
    Console.WriteLine( "OOPS! The objects are the same!" );
else
    Console.WriteLine( "The objects are not the same!" );
```

Output

The objects are identical copies!

The objects are not the same!

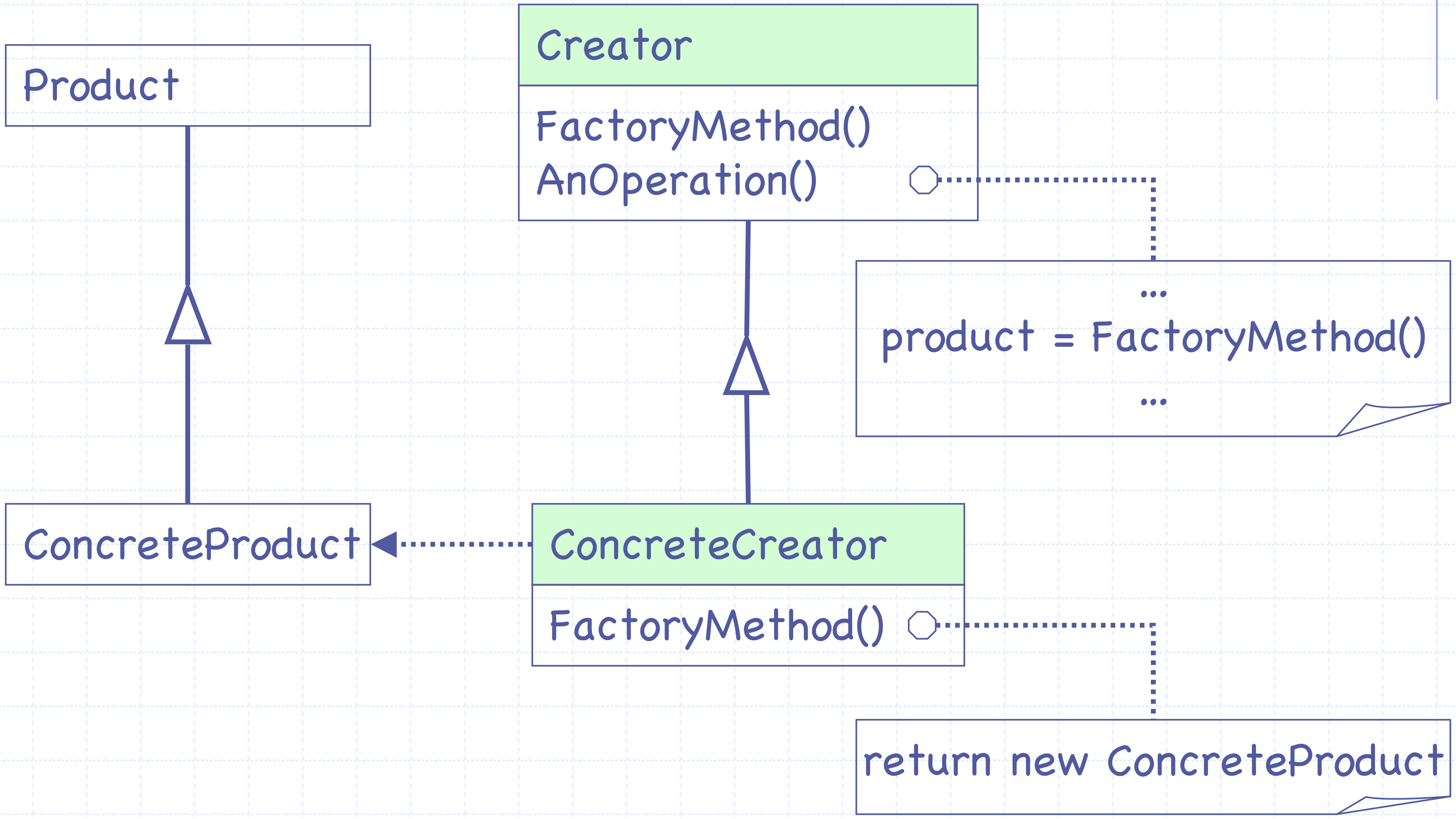


Factory Method

- Intent:
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- Collaborations:
  - Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate *ConcreteProduct*.

Structure of Factory Method



Classical Example

- A classical example of factory method is that of iterators.
- An iterator provides access to elements of a collection. A concrete iterator methods isolate the caller from knowing, which class to instantiate.

HuffmanBitCode::begin()

```
HuffmanBitCode::iterator HuffmanBitCode::begin()
{
    return iterator( this );
}
```

HuffmanBitCode::end()

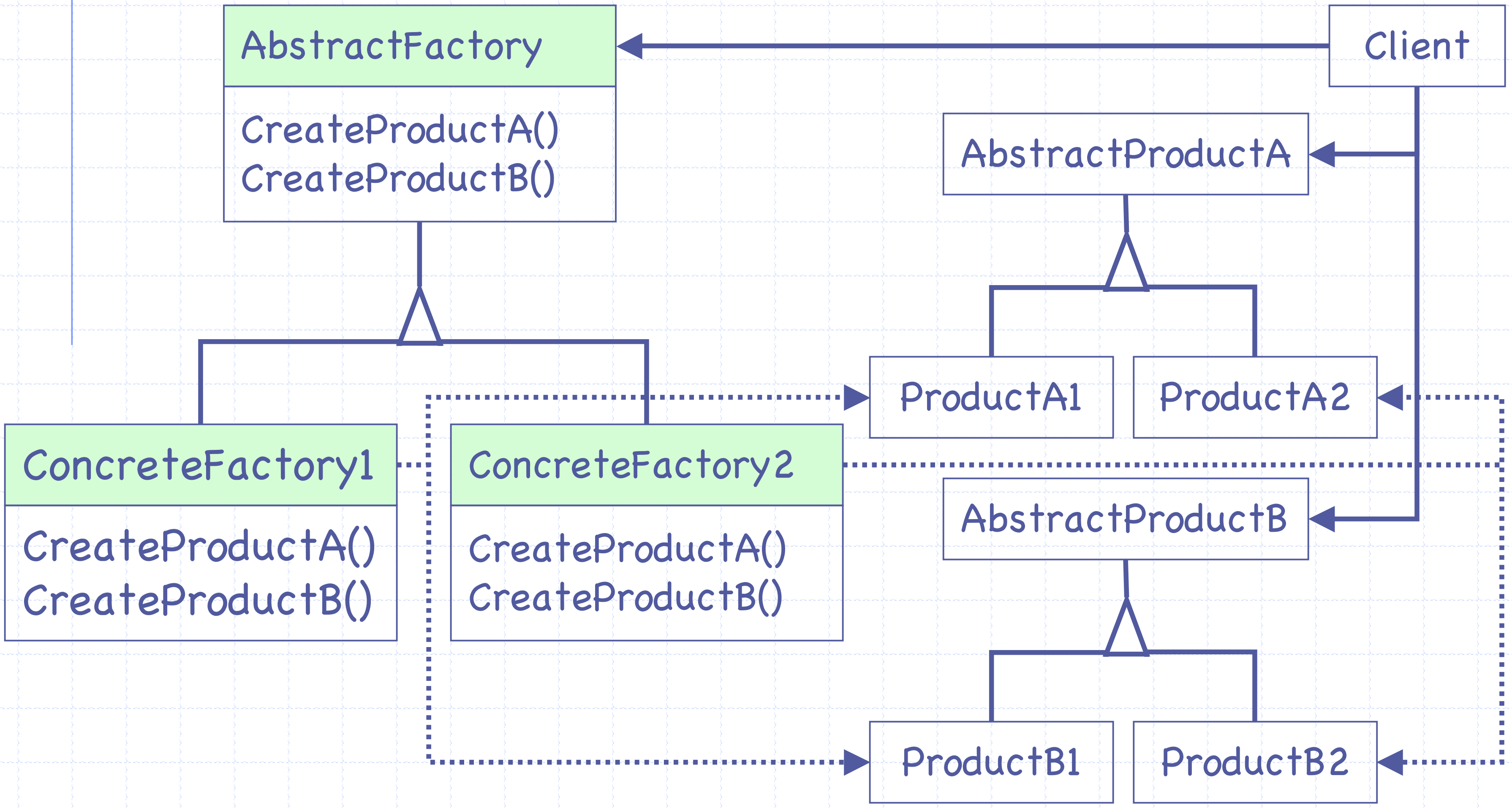
```
HuffmanBitCode::iterator HuffmanBitCode::end()
{
    return iterator( this, fCodeLength );
}
```

Abstract Factory

- Intent:
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- Collaborations:
  - Normally a single instance of a ConcreteFactory class is created at runtime. This concrete factory creates product objects having a particular implementation.
  - To create different product objects, clients should use a different concrete factory.
  - AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

Structure of Abstract Factory



Skeleton Code

```
class ProductClass
{
public:
    ProductClass() { … }
    ProductClass( int aInitValue ) { … }
};

class FactoryClass
{
public:
    ProductClass* GetNewInstance()
            { return new ProductClass(); }
    ProductClass* GetNewInstance( int aInitValue )
            { return new ProductClass( aInitValue ); }
};
```

Abstract Factory Client

```
class ClientClass
{
public:
    void start()
    {
        // create a new factory
        ProductFactory lFactory;

        // create objects
        ProductClass* lObj1 = lFactory.GetNewInstance();
        ProductClass* lObj2 = lFactory.GetNewInstance( 4 );
    }
};
```

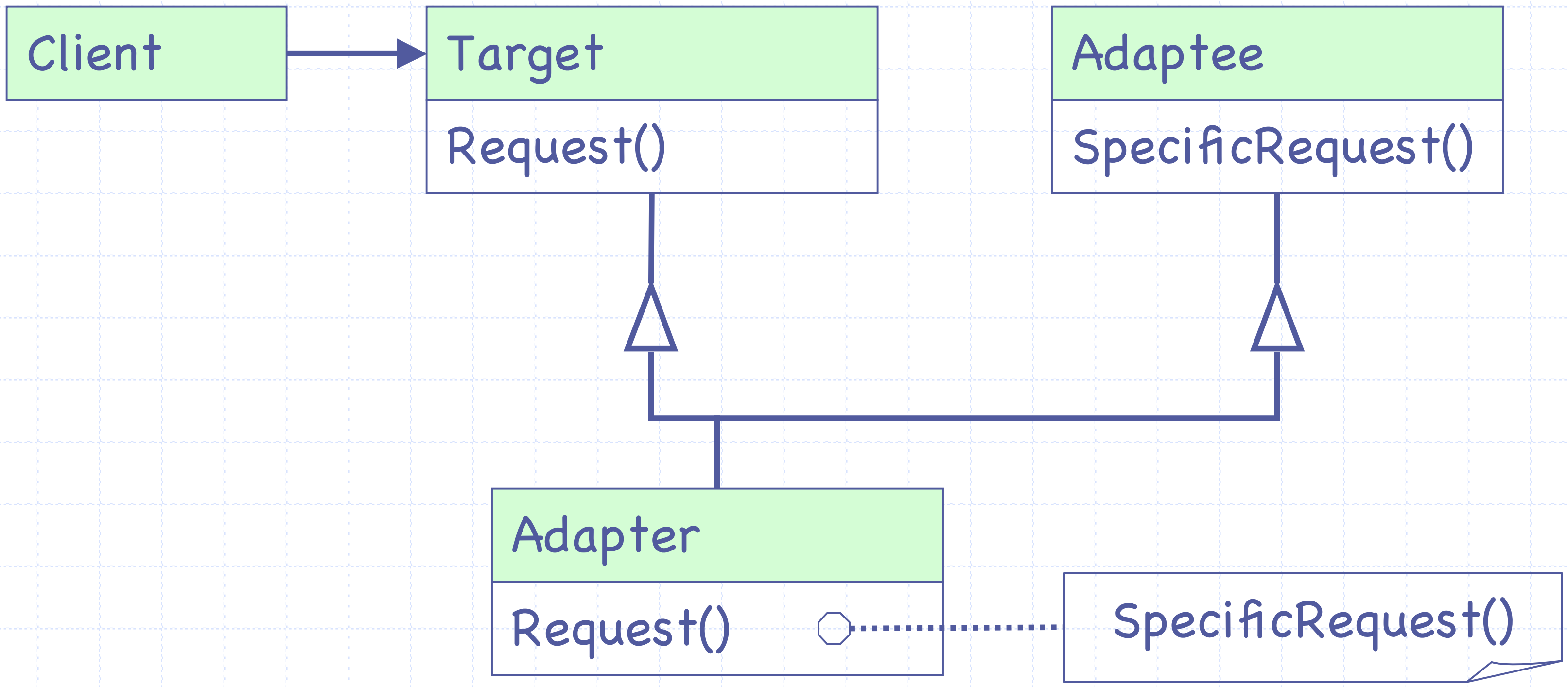
Structural Patterns

- Structural patterns are concerned with how classes and object are composed to form larger structures.
  - Structural *class* patterns use inheritance to compose interfaces or implementations.
  - Structural *object* patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition.

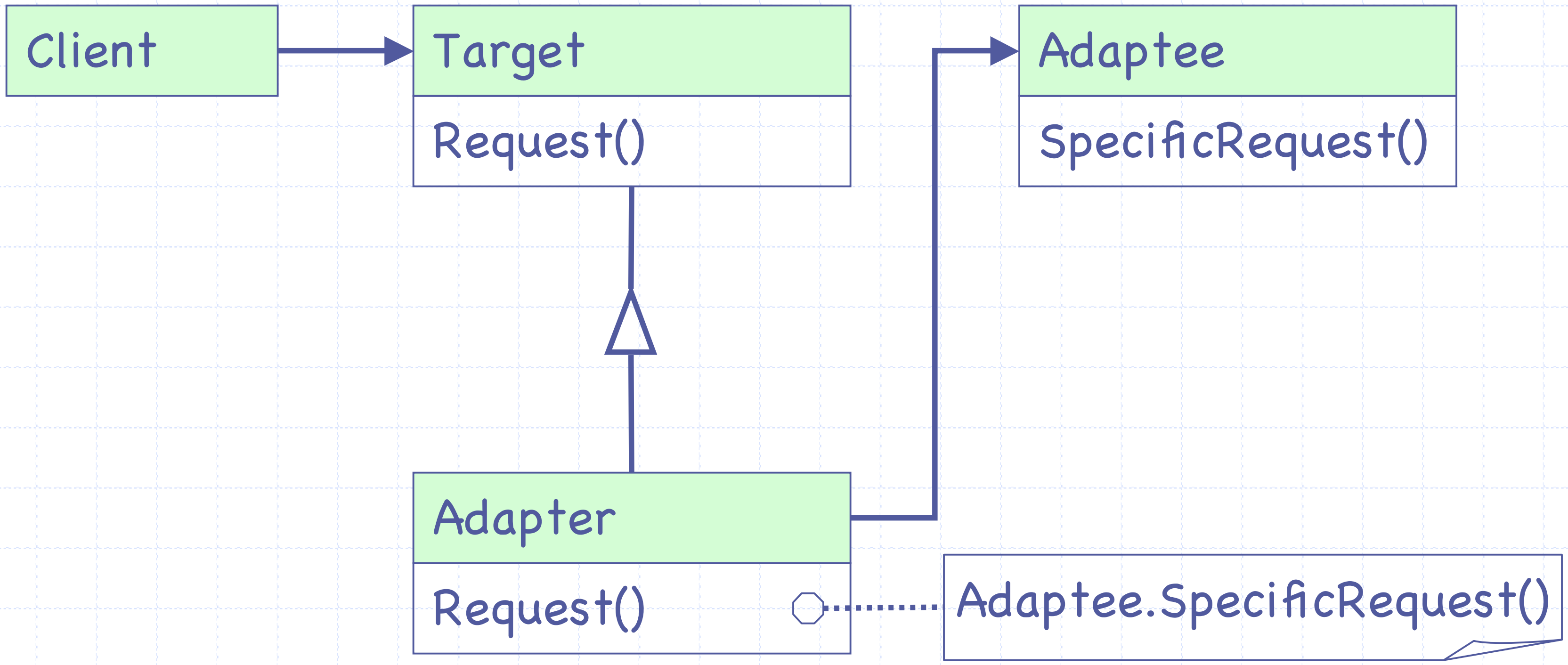
Adapter

- Intent:
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.
- Collaborations:
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Structure of a Class Adapter



Structure of an Object Adapter



Class Adapter: Target

```
class ITower
{
    string getName() = 0;
    void add( Disk& aDisk ) = 0;
    Disk remove() = 0;
};
```

Problem:

We want to use stack<T> to implement the class Tower to solve the problem “Towers of Hanoi”.

Class Adapter: Adapter

```
class Tower : public stack<Disk>, ITower
{
private:
    string fName;

public:
    Tower( string aName ) : stack<Disk>() { fName = aName; }

    string getName() { return fName; }       // ITower.Name
    void add( Disk& aDisk )                  // ITower.Add
    {
        if ( (size() > 0) && (aDisk.compareTo( top() ) > 0) )
            throw IllegalSizeException( … );
        else
            push( aDisk ); }                 // SpecificRequest()
    public Disk remove()                     // ITower.Remove
    { Disk Result = top(); pop(); return Result; } // SpecificRequest()
};
```

Object Adapter: Target

```
class ITower2
{
    int getCount() = 0;
    string getName() = 0;
    void add( Disk& aDisk ) = 0;
    Disk remove() = 0;
};
```

We will use a stack<T> instance variable and forward calls of add and remove to methods of stack<T>.

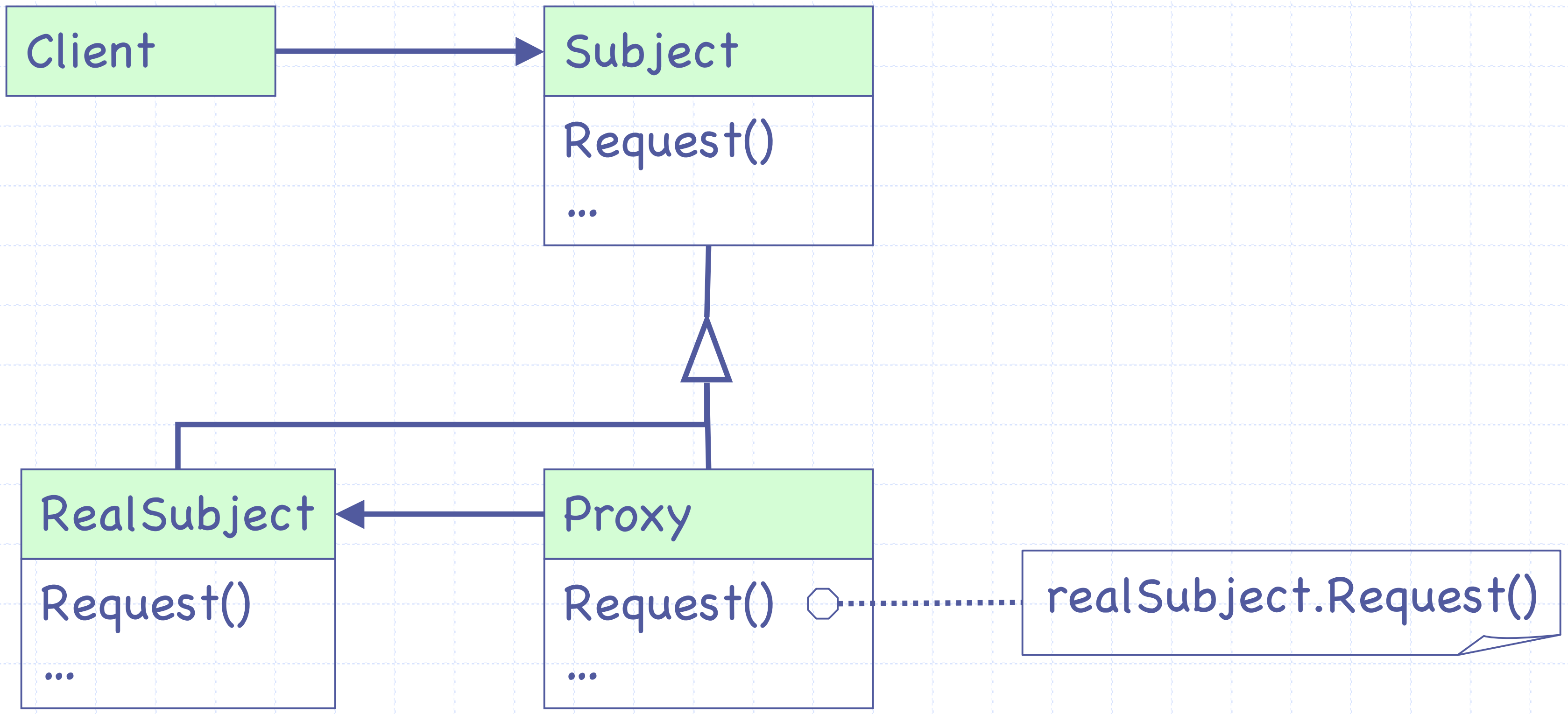
Object Adapter: Adapter

```
class Tower : public ITower2
{
private:
    string fName;
    stack<Disk> fStack; // delegate instance

public:
    Tower( string aName ) { fName = aName; }

    int getCount() { return fStack.size(); }   // ITower2.Count
    string getName() { return fName;}          // ITower2.Name
    void add( Disk& aDisk )                    // ITower2.Add
    {   if ( (getCount() > 0) && (aDisk.compareTo( fStack.top() ) > 0) )
            throw IllegalDiskSizeException( … );
        else fStack.push( aObject ); }
    Disk remove()                              // ITower2.Remove
    { Disk Result = fStack.top(); fStack.pop(); return Result; }
}
```

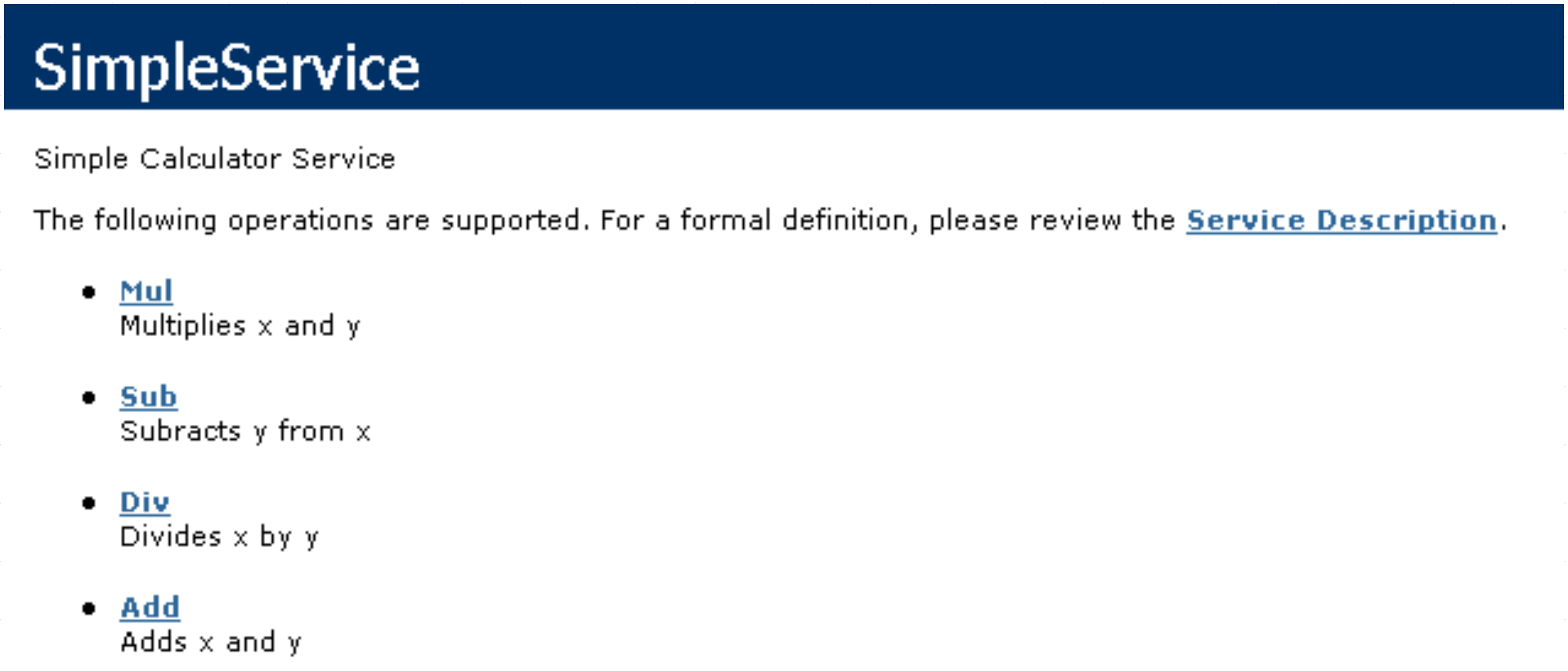
Proxy

- Intent:
  - Provide a surrogate or placeholder for another object to control access to it.
- Collaborations:
  - Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Structure of Proxy



Calculator Web Service

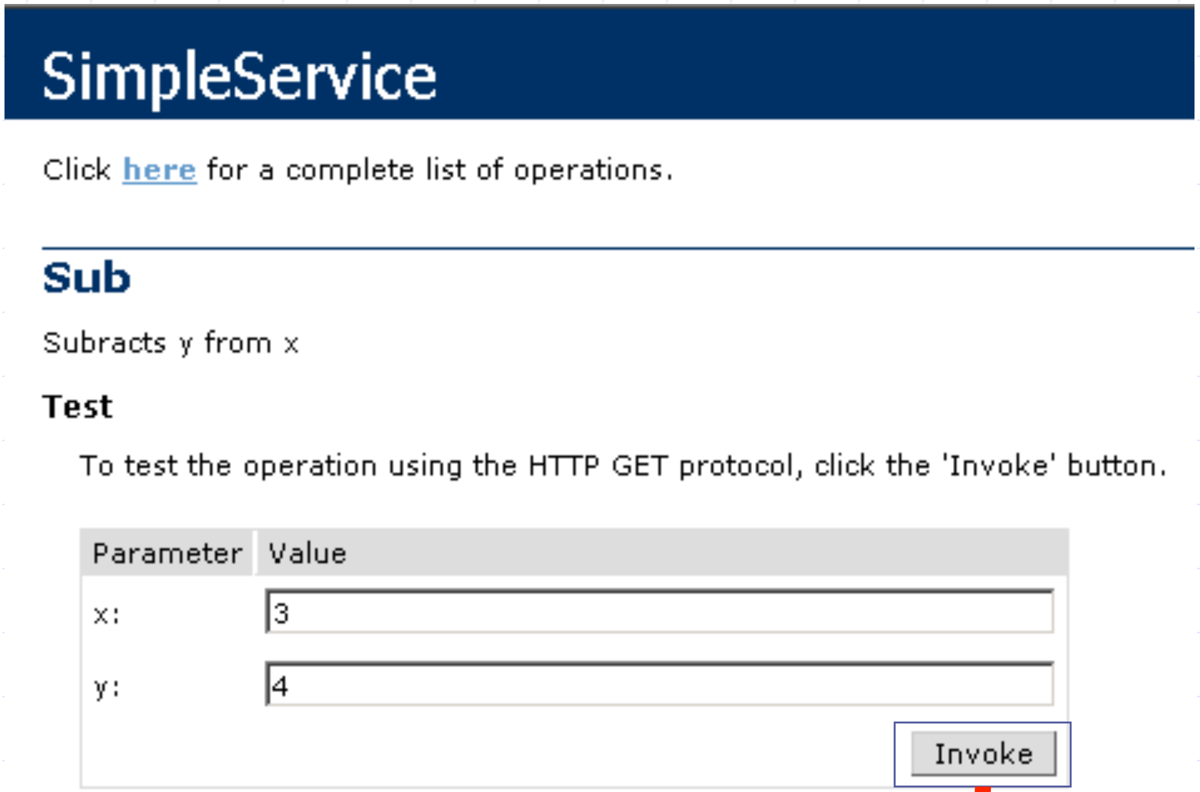

SimpleService

Simple Calculator Service

The following operations are supported. For a formal definition, please review the Service Description.

- Mul
  Multiplies x and y
- Sub
  Subracts y from x
- Div
  Divides x by y
- Add
  Adds x and y

…/Service1.asmx/Sub?x=3&y=4



SimpleService

Click here for a complete list of operations.

Sub

Subracts y from x

Test

To test the operation using the HTTP GET protocol, click the 'Invoke' button.

Parameter	Value
x:	3
y:	4

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://localhost/">-1</int>
```

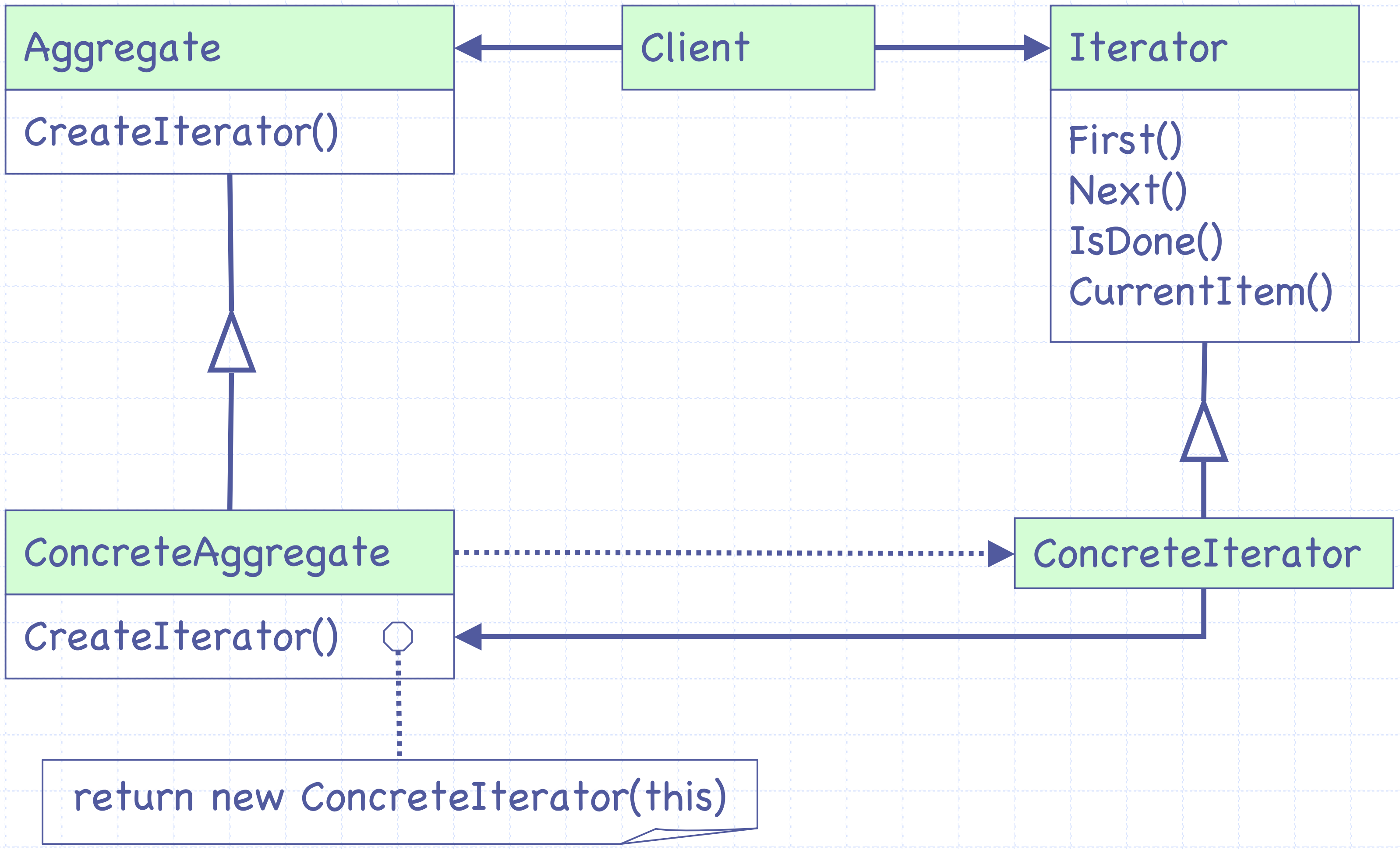
Behavioral Patterns

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
  - Behavioral *class* patterns use inheritance to distribute behavior between classes.
  - Behavioral *object* patterns object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.
- The classic example of a behavioral pattern is Model-View-Controller (Observer), where all views of the model are notified whenever the model's state changes.

Iterator

- Intent:
  - Provide a way to access the elements of n aggregate object sequentially without exposing its underlying representation.
- Collaborations:
  - A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

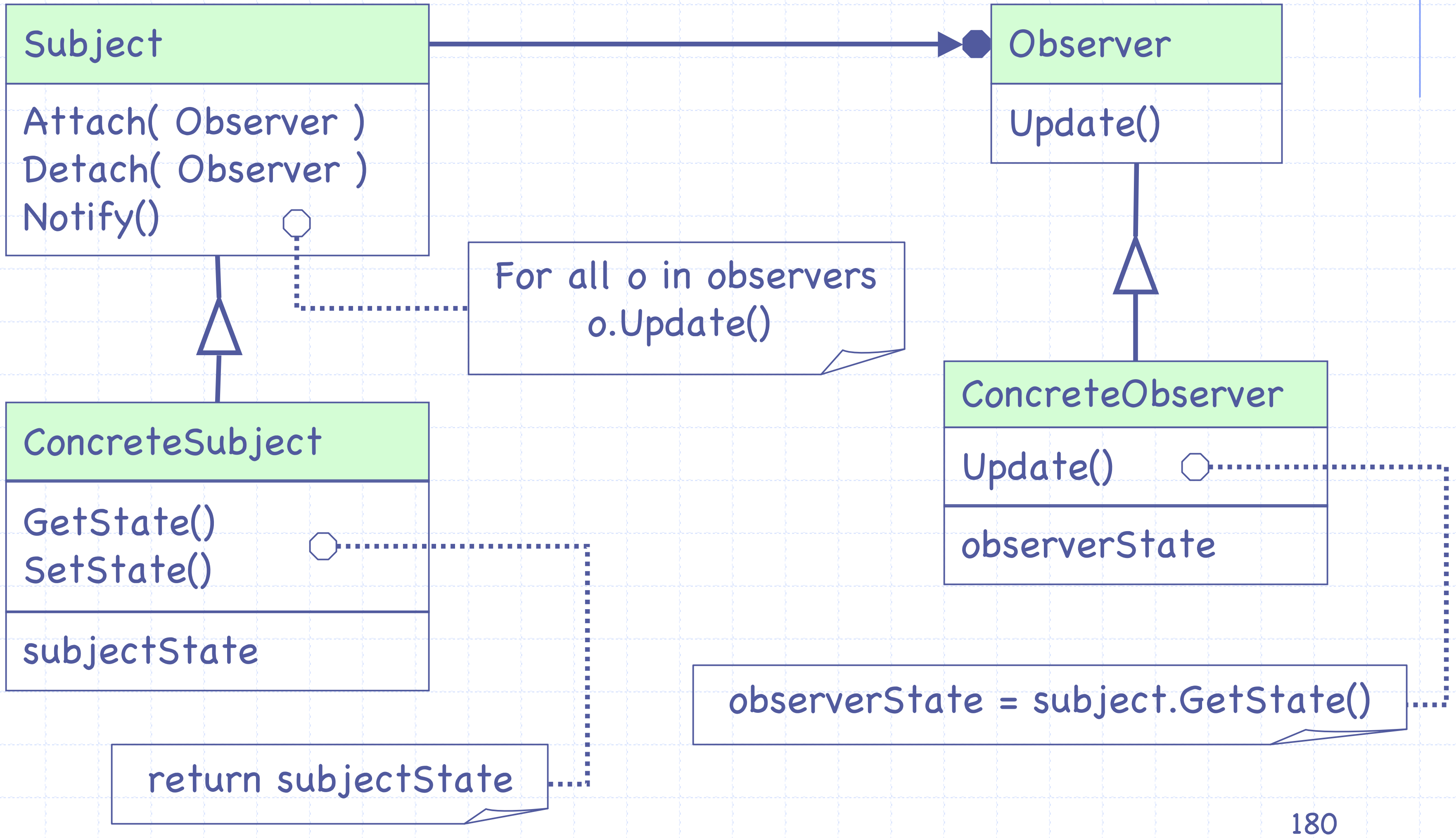
Structure of Iterator



Observer

- Intent:
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Collaborations:
  - ConcreteSubject notifies its observers whenever a change occurs that could make its observer's state inconsistent with its own.

Structure of Observer



Iterator Reviewed

- An iterator remains valid as long as the collection remains unchanged.
- If changes are made to the collection, such as adding, modifying or deleting elements, the iterator is irrecoverably invalidated and any access to its underlying elements must throw an exception.

Class HuffmanBitCode

```
class HuffmanBitCode
{
private:
    byte* fBitArray;
    unsigned int fCodeLength;
    iterator fIterator;

    void notify();
    void copy( const HuffmanBitCode& aCode );
    void extend( int aToCodeLength );
    void copy( byte aBitArray[], int aCodeLength );

    …
};
```

Class HuffmanBitCode::iterator

```
class iterator
{
private:
    bool fCollectionChanged;  // new private variable

public:
    void changedCollection()   // Update()
    { fCollectionChanged = true; }

    int iterator::operator*() const
    {
      if ( !CollectionChanged && fIndex < fCode->size() )
              return (*fCode)[fIndex];
      else
         throw HuffmanException( "Invalid iterator!" );
    }
...
};
```