

Môn học

PHÂN TÍCH & THIẾT KẾ HƯỚNG ĐỐI TƯỢNG



Tài liệu tham khảo chính

- [1] **The Unified Software Development Process**, Ivar Jacobson, Grady Booch, James Rumbaugh, Addison-Wesley, 1999.
- [2] **Software Engineering - A practitioner's approach**, R.S. Pressman, McGraw-Hill, 1997
- [3] **Design Patterns**, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1998.
- [4] OMG Unified Modeling Language Specification, version 1.3, Object Management Group (www.omg.org), 1999
- [5] UML Toolkit, Hans-Erik Eriksson & Magnus Penker, 1998
- [6] Object-Oriented Software Engineering, A Use-Case Driven Approach, I. Jacobson, ACM Press/Addison-Wesley, 1992
- [7] Object-Oriented Analysis and Design with Applications, G. Booch, The Benjamin Cummings Publishing Company, 1994



Chương 1

CÁC KHÁI NIỆM CƠ BẢN CỦA MÔ HÌNH HƯỚNG ĐỐI TƯỢNG



Nội dung

1.1 Từ lập trình có cấu trúc đến OOP

1.2 Đối tượng, thuộc tính, tác vụ.

1.3 Abstract type và class.

1.4 Tính bao đóng.

1.5 Tính thừa kế và cơ chế '*override*'.

1.6 Tính bao gộp.

1.7 Thông điệp, tính đa hình và kiểm tra kiểu.

1.8 Tính tổng quát hóa.

1.9 Tính vững bền.



Từ lập trình có cấu trúc đến OOP

1. **Máy tính số** là thiết bị có thể thực hiện 1 số hữu hạn các chức năng cơ bản (**tập lệnh**), cơ chế thực hiện các lệnh là **tự động từ lệnh đầu cho đến lệnh cuối cùng**. Danh sách các lệnh được thực hiện này được gọi là **chương trình**.
2. bất kỳ công việc ngoài đời nào cũng có thể được chia thành trình tự nhiều công việc nhỏ hơn. Trình tự các công việc nhỏ này được gọi là **giải thuật** giải quyết công việc ngoài đời. Mỗi công việc nhỏ hơn cũng có thể được chia nhỏ nữa,... \Rightarrow công việc ngoài đời là 1 trình tự các lệnh máy (**chương trình**).
3. vấn đề mấu chốt của việc dùng máy tính giải quyết vấn đề ngoài đời là **lập trình**. Cho đến nay, **lập trình là công việc của con người** (với sự trợ giúp ngày càng nhiều của máy tính).
4. các lệnh của chương trình (code) phải tham khảo hoặc xử lý (truy xuất) **thông tin (dữ liệu)**.



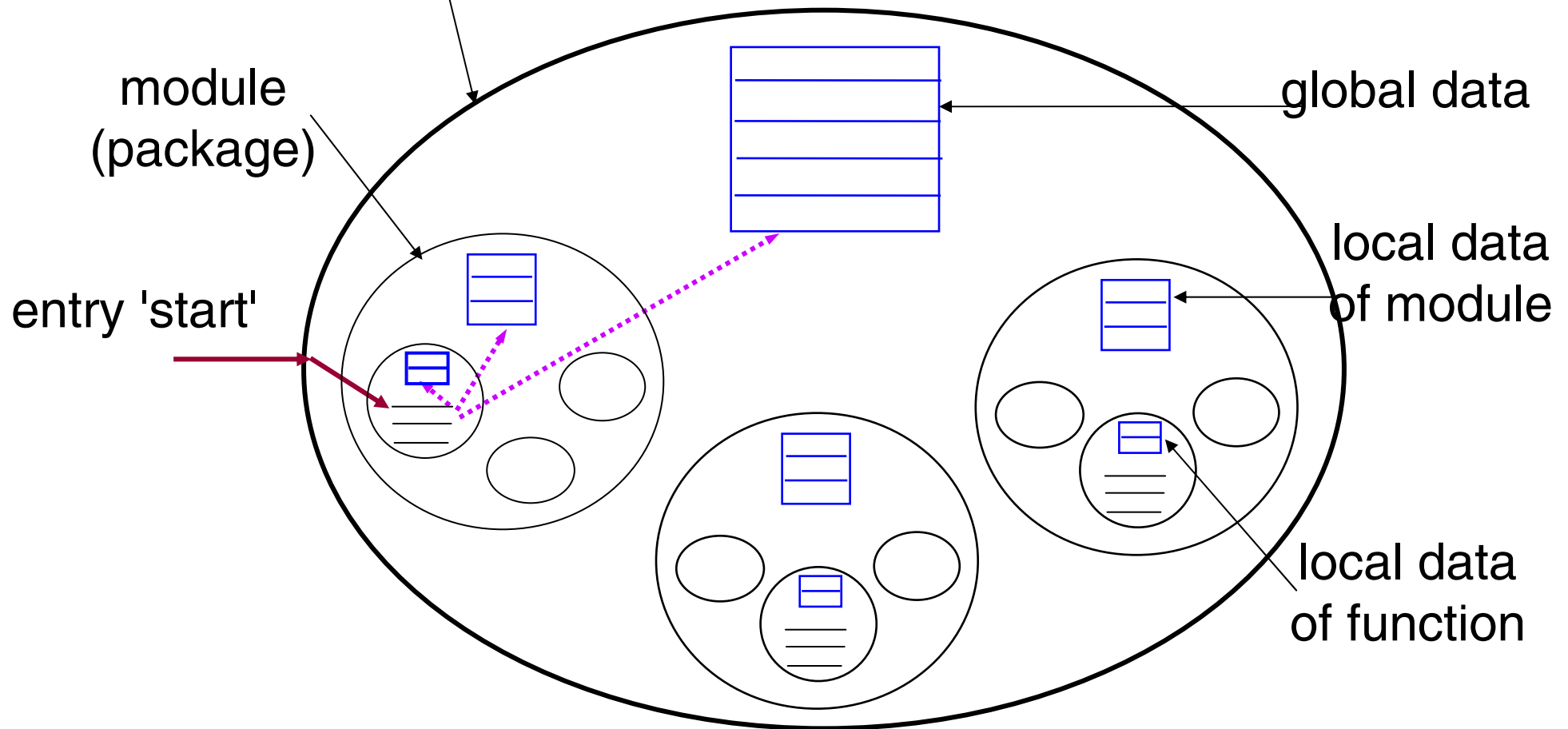
Từ lập trình có cấu trúc đến OOP

5. Dữ liệu của 1 chương trình có thể rất nhiều và đa dạng. Để truy xuất đúng 1 dữ liệu ta cần :
 - tên nhận dạng.
 - kiểu dữ liệu miêu tả cấu trúc dữ liệu.
 - tầm vực truy xuất miêu tả giới hạn khách hàng truy xuất dữ liệu.
6. Chương trình cổ điển = giải thuật + dữ liệu.
7. Chương trình con (function, subroutine,...) cho phép cấu trúc chương trình, sử dụng lại code...
8. Chương trình cổ điển có cấu trúc phân cấp như sau :



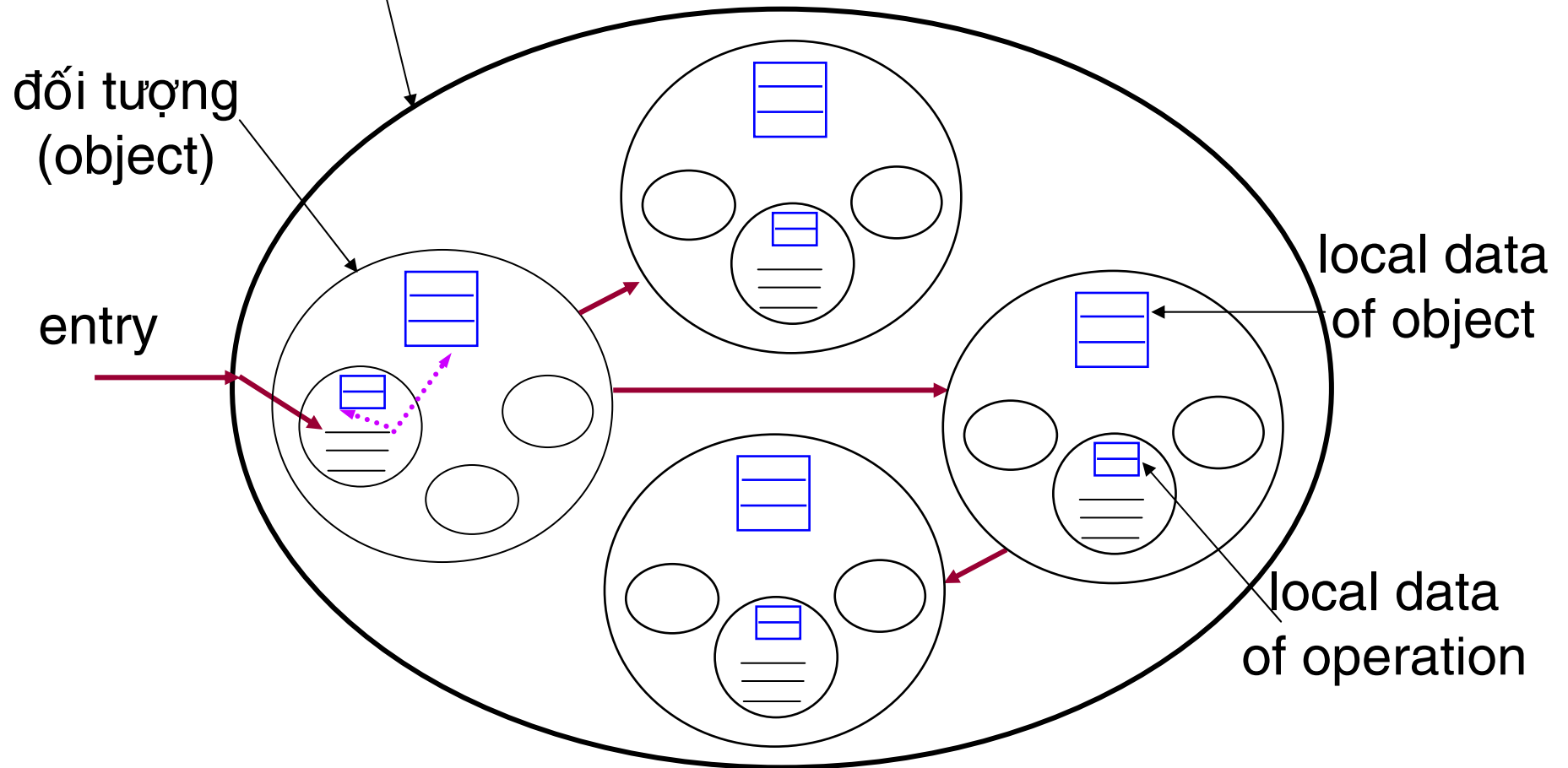
Từ lập trình có cấu trúc đến OOP

Chương trình = cấu trúc dữ liệu + giải thuật



Từ lập trình có cấu trúc đến OOP

Chương trình = tập các đối tượng tương tác nhau



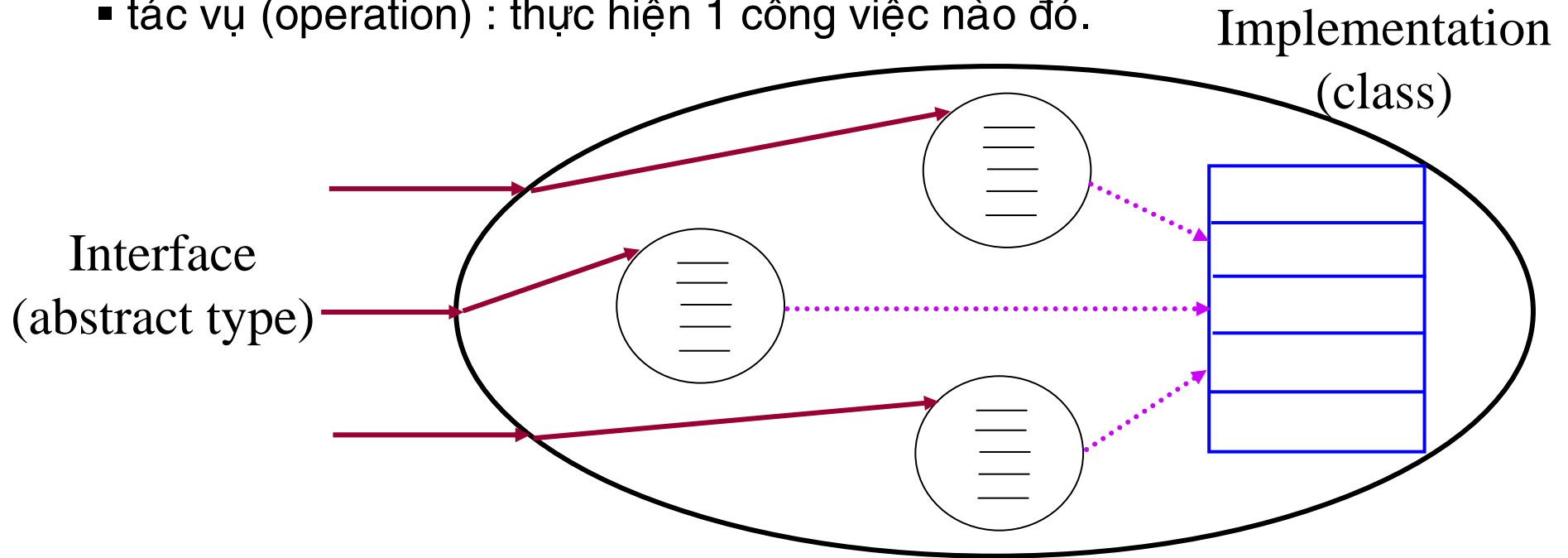
Tổng quát về hướng đối tượng

- Mô hình hướng đối tượng giới thiệu 1 quan điểm lập trình (và phân tích/thiết kế) khác hẳn so với trường phái cổ điển (có cấu trúc).
- Bắt đầu nhen nhóm vào những năm cuối 60s và đến đầu 90s thì trở nên rất phổ biến trong công nghiệp phần mềm.
- Những ngôn ngữ hướng đối tượng đầu tiên : Smalltalk, Eiffel. Sau đó xuất hiện thêm : Object Pascal, C++, Java, C#,...
- Hình thành các phương pháp phân tích/thiết kế hướng đối tượng.
- Và hiện nay ta có 1 quy trình phát triển phần mềm hợp nhất dựa trên ngôn ngữ UML.



Đối tượng (Object)

- ☉ Mô hình đối tượng quan niệm chương trình bao gồm các đối tượng sinh sống và tương tác với nhau.
- ☉ Đối tượng bao gồm :
 - thuộc tính (dữ liệu) : mang 1 giá trị nhất định tại từng thời điểm.
 - tác vụ (operation) : thực hiện 1 công việc nào đó.



Kiểu trừu tượng (Abstract type)

- ⊙ **Abstract type** (*type*) định nghĩa interface sử dụng đối tượng.
- ⊙ **Interface** là tập các entry mà bên ngoài có thể giao tiếp với đối tượng.
- ⊙ Dùng **signature** để định nghĩa mỗi entry, Signature gồm :
 - **tên** method (operation)
 - **danh sách đối số** hình thức, mỗi đối số được đặc tả bởi 3 thuộc tính : tên, type và chiều chuyển động (IN, OUT, INOUT).
 - **đặc tả chức năng** của method (thường là chú thích).
- ⊙ Dùng abstract type (chứ không phải class) để đặc tả kiểu cho biến, thuộc tính, tham số hình thức.
- ⊙ User không cần quan tâm đến class (hiện thực cụ thể) của đối tượng.

Class (Implementation)

- ⊙ Class định nghĩa chi tiết hiện thực đối tượng :
 - định nghĩa các thuộc tính dữ liệu : giá trị của tất cả thuộc tính xác định trạng thái của đối tượng.
 - kiểu của thuộc tính có thể là type cổ điển hay abstract type, trong trường hợp sau thuộc tính chứa tham khảo đến đối tượng khác.
 - coding các method và các internal function.
- ⊙ Định nghĩa các method tạo và xóa đối tượng.
- ⊙ Định nghĩa các method constructor và destructor.
- ⊙ User không cần quan tâm đến class của đối tượng.

Ví dụ về class trong Java

```
class abstract HTMLObject {
    protected static final int LEFT = 0;
    protected static final int MIDDLE = 1;
    protected static final int RIGHT = 2;
    private int alignment = LEFT;
    protected Vector objects = null;
    HTMLObject( ) {           // constructor
        objects = new Vector (5);
    }
    public void setAlignment( int algnmt ) {
        alignment = algnmt;
    }
    public int getAlignment( ) {
        return alignment;
    }
    public abstract String toHTML( ); // abstract operation
}
```



Tính bao đóng (encapsulation)

- Bao đóng : che dấu mọi chi tiết hiện thực của đối tượng, không cho bên ngoài thấy và truy xuất \Rightarrow tính độc lập cao giữa các đối tượng (hay tính nối kết - coupling giữa các đối tượng rất thấp).
 - che dấu các thuộc tính dữ liệu : nếu cần cho phép truy xuất 1 thuộc tính dữ liệu, ta tạo 2 method get/set tương ứng để giám sát việc truy xuất và che dấu chi tiết hiện thực bên trong.
 - che dấu chi tiết hiện thực các method.
 - che dấu các internal function và sự hiện thực của chúng.

Tính thừa kế (inheritance)

- ◎ Tính thừa kế cho phép giảm nhẹ công sức định nghĩa type/class : ta có thể định nghĩa các type/class không phải từ đầu mà bằng cách kế thừa type/class có sẵn, ta chỉ định nghĩa thêm các chi tiết mới mà thôi (thường khá ít).
 - Đa thừa kế hay đơn thừa kế.
 - Mỗi quan hệ supertype/subtype và superclass/subclass.
 - có thể override các method của class cha, kết quả override chỉ có nghĩa trong đối tượng class con.
 - Đối tượng của class con có thể đóng vai trò của đối tượng cha nhưng ngược lại thường không đúng.

Ví dụ về thừa kế và override - Java

```
class Geometry {  
    public Draw(Graphics g);  
    protected int      xPos, yPos;  
    protected double    xScale, yScale;  
    protected COLORREF  color;  
};
```

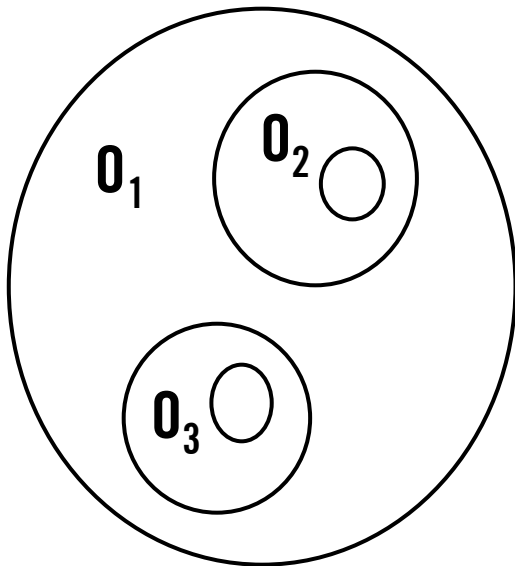
```
class Line extends Geometry {  
    int xPos2, yPos2;  
    // other attributes...  
    public Draw(Graphics g) {  
        // các lệnh vẽ đoạn thẳng  
    }  
}
```



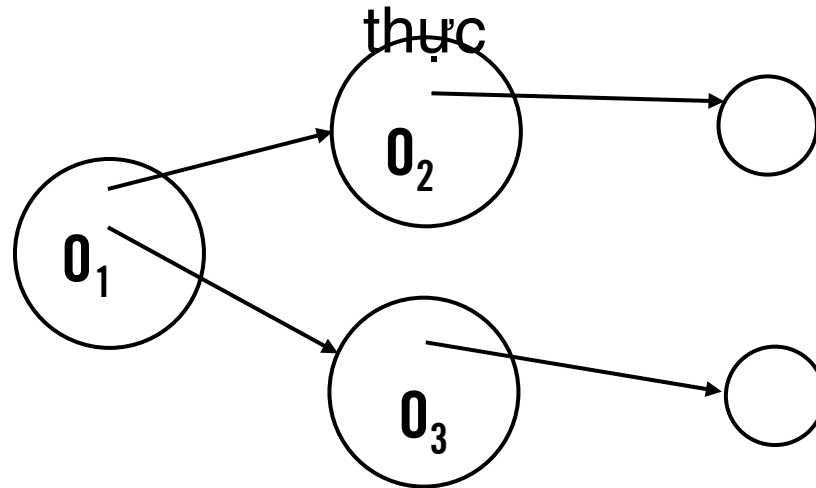
Tính bao gộp (aggregation)

- 1 đối tượng có thể chứa nhiều đối tượng khác tạo nên mối quan hệ bao gộp 1 cách đệ qui giữa các đối tượng.
- Có 2 góc nhìn về tính bao gộp : ngữ nghĩa và hiện thực.

Góc nhìn ngữ nghĩa



Góc nhìn hiện thực



Ví dụ về bao gộp - C++

```
class Geometry {                                // abstract base class
public:
    Geometry( );
    ~Geometry( );
    virtual void Draw( Window *pWnd ) = 0; // abstract operation
protected:
    int      xPos, yPos;
    double   xScale, yScale;
    COLORREF color;
};

class Group : public Geometry {
public:
    Group( );
    ~Group( );
    virtual void Draw( Window *pWnd ); // override
private:
    Geometry **ppGeo;    // pointer container
    int      geoCount;
};
```



Thông điệp (Message)

- ⊙ Thông điệp là 1 phép gọi tác vụ đến 1 đối tượng từ 1 tham khảo.
- ⊙ Thông điệp bao gồm 3 phần :
 - **tham khảo** đến đối tượng đích.
 - **tên tác vụ** muốn gọi.
 - **danh sách tham số thực** cần truyền theo (hay nhận về từ) tác vụ.
 - ví dụ : `aCircle.SetRadius (3);` `aCircle.Draw (pWnd);`
- ⊙ Thông điệp là phương tiện giao tiếp (hay tương tác) duy nhất giữa các đối tượng.

Tính đa xạ (Polymorphism)

- ⊙ Cùng 1 lệnh gọi thông điệp đến đối tượng thông qua cùng 1 tham khảo nhưng ở vị trí/thời điểm khác nhau có thể gây ra việc thực thi method khác nhau của các đối tượng khác nhau.

T1 p1; // C1 và C2 là 2 class hiện thực T1

...

p1 = New C1; // tạo đối tượng C1, gán tham khảo vào p1
p1.meth1(...);

...

p1 = New C2; // tạo đối tượng C2, gán tham khảo vào p1
p1.meth1(...);

Lệnh **p1.meth1(...)**; ở 2 vị trí khác nhau kích hoạt 2 method khác nhau của 2 class khác nhau.



Kiểm tra kiểu (type check)

- ⊙ Chặt và dùng mối quan hệ 'conformity' (tương thích tổng quát). Type A tương thích với type B \Leftrightarrow A chứa mọi method của B và ứng với từng method của B :
 - tồn tại 1 method cùng tên trong A.
 - danh sách đối số của 2 method tương ứng phải bằng nhau.
 - kiểu đối số OUT hay giá trị return của method trong A phải tương thích với kiểu của đối số tương ứng trong B.
 - kiểu đối số IN của method trong B phải tương thích với kiểu của đối số tương ứng trong A.
 - kiểu đối số INOUT của method trong A phải trùng với kiểu của đối số tương ứng trong B.
- ⇒ quan hệ so trùng hay quan hệ con/cha (sub/super) là trường hợp đặc biệt của quan hệ tương thích tổng quát.

Tính tổng quát hóa (Generalization)

- ☉ Có 2 ngữ nghĩa khác nhau của tính tổng quát hóa :
 - class tổng quát hóa cho phép **sản sinh tự động các class** bình thường, các class bình thường tự nó chỉ có thể tạo ra đối tượng. Thường dùng ngữ nghĩa này trong giai đoạn lập trình.
 - **ngược với tính thừa kế** : supertype/superclass là type/class tổng quát hóa của các con của nó. Thường dùng ngữ nghĩa này trong giai đoạn phân tích/thiết kế phần mềm.

Tính thường trú (persistence)

- ⊙ đời sống của 1 đối tượng độc lập với đời sống của phần tử tạo ra nó.
 - **đối tượng phải tồn tại** khi còn ít nhất 1 tham khảo đến nó trong hệ thống.
 - **đối tượng phải bị xóa** khi không còn tham khảo nào đến nó, vì tại thời điểm này đối tượng là rác. Việc xác định chính xác 1 đối tượng có phải là rác hay không là 1 việc phức tạp code ứng dụng không được phép làm, đây là công việc của hệ thống thông qua module 'garbage collection'.
 - **vững bền không phải là vĩnh hằng**, mức độ có thể là 1 session của máy ảo (JVM) hay lâu dài (thông qua đĩa cứng, CDROM).

Tổng kết

- ◎ Mô hình hướng đối tượng quan niệm thế giới (hay chương trình) bao gồm các đối tượng sống chung và tương tác với nhau.
- ◎ Các đặc điểm chính của hướng đối tượng :
 - Bao đóng : mỗi đối tượng bao gồm dữ liệu và tác vụ. Các tác vụ thiết lập nên hành vi của đối tượng. Các đối tượng được phân loại bằng class.
 - Các đối tượng tương tác với nhau bằng cách gọi thông điệp.
 - giữa các class/đối tượng có thể tồn tại quan hệ bao gộp, thừa kế, tổng quát hóa.
 - Tính đa hình : kết quả của sự kiểm tra kiểu dựa vào mối quan hệ 'conformity'.
 - Tính vững bền : đối tượng tồn tại khi còn ít nhất 1 tham khảo đến nó.

Chương 2

THÍ DỤ VỀ NGÔN NGỮ OOP

👉 Visual C++

👉 Java

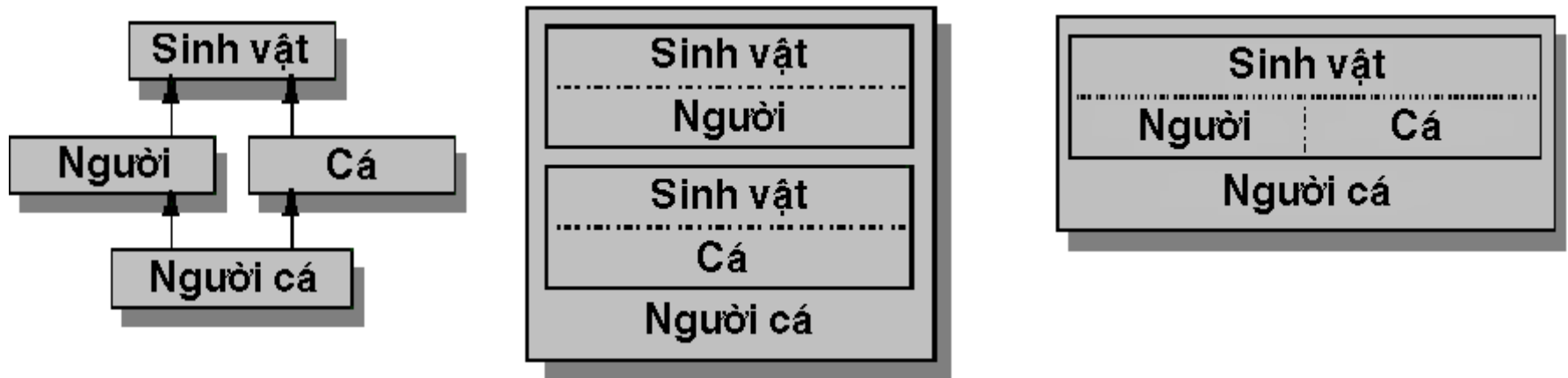
2.1 Ngôn ngữ Visual C++

1. Chỉ hỗ trợ khái niệm class.
2. Cho phép Đa thừa kế.
3. Dùng 'abstract class' để định nghĩa interface.
4. Tầm vực truy xuất các thành phần.
5. Đa hình có chọn lọc nhờ 'virtual function'
6. Chỉ hỗ trợ các đối tượng tạm.
7. Override method khi thừa kế.
8. Có thể định nghĩa function overloaded.



Chỉ hỗ trợ khái niệm class


1. Dùng class để định nghĩa kiểu cho các biến, thuộc tính
⇒ đối tượng có thể chứa vật lý đối tượng khác hay chứa tham khảo đến đối tượng khác.
2. Đa thừa kế trong định nghĩa class ⇒ 1 class có thể chứa nhiều class cha trùng nhau ⇒ dùng "virtual base class" để tối ưu hóa bộ nhớ đối tượng.



Class trừu tượng (Abstract class)

3. Hỗ trợ khái niệm "abstract class" để định nghĩa class chỉ chứa thông tin interface nhưng không cho phép dùng class này để định nghĩa kiểu cho biến hay thuộc tính. 1 abstract class là 1 class chứa ít nhất 1 "pure virtual function".

```
class Geometry {           // abstract class
public:
    Geometry( );
    ~Geometry( );
    virtual void Draw( Window *pWnd ) = 0; // pure virtual function
protected:
    int          xPos, yPos;
    double       xScale, yScale;
    COLORREF     color;
};
```



Tầm vực truy xuất thành viên

4. Tầm vực truy xuất thông tin trong đối tượng :

private : thông tin bị che dấu hoàn toàn.

protected : chỉ che dấu bên ngoài nhưng cho phép các đối tượng con, cháu, chắt... truy xuất.

public : cho phép tất cả mọi nơi truy xuất.

Friend class : là class mà mỗi function của nó đều có thể truy xuất tự do mỗi thành phần của class hiện tại.

Friend function : là function có thể truy xuất tự do mỗi thành phần của class hiện tại.

Có thể hạn chế tầm vực của thành viên của class cha khi thừa kế.

Hỗ trợ tính đa hình có chọn lọc

5. Định nghĩa 'virtual function' nếu muốn áp dụng tính đa hình trong việc gọi thông báo yêu cầu function này thực thi. Tất cả các 'virtual function' được quản lý trong 1 danh sách "virtual function table".

địa chỉ function 1
địa chỉ function 2
địa chỉ function 3
địa chỉ function i
địa chỉ function n

Các đối tượng đều tạm thời

6. Các đối tượng chỉ tồn tại tạm thời trong không gian process.
Tham khảo đến đối tượng thực chất là **pointer** cục bộ.
chương trình phải tự viết code cho hoạt động save/restore đối tượng nếu muốn lưu giữ/dùng lại đối tượng.
VC++ hỗ trợ hoạt động save/restore đối tượng nhờ khả năng 'Serialization'.
7. Có quyền '**override**' bất kỳ toán tử hay function nào của class cha.
8. Cho phép định nghĩa các hàm '**overloaded**' : cùng tên nhưng 'signature' khác nhau.

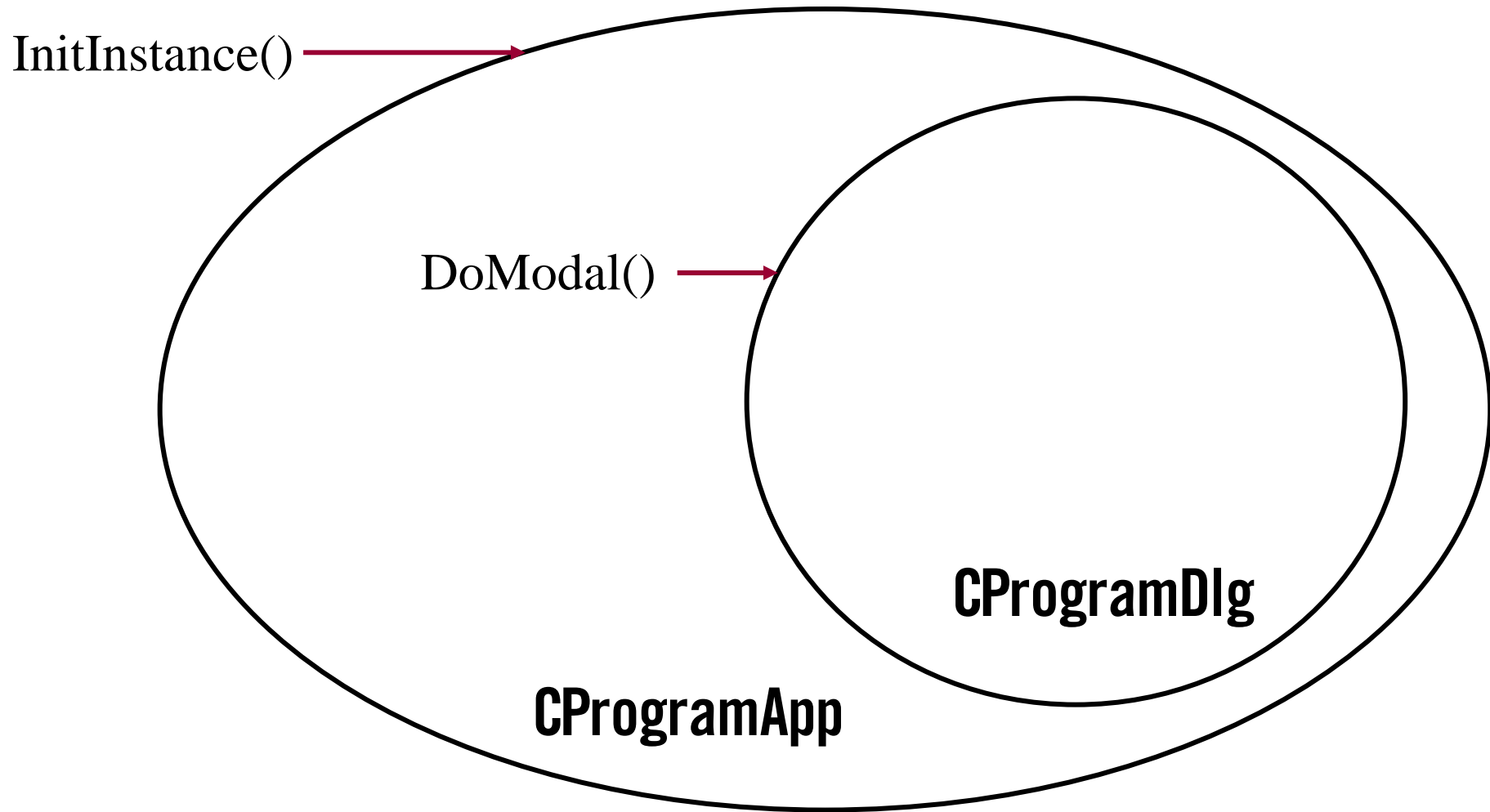


Skeleton định nghĩa class

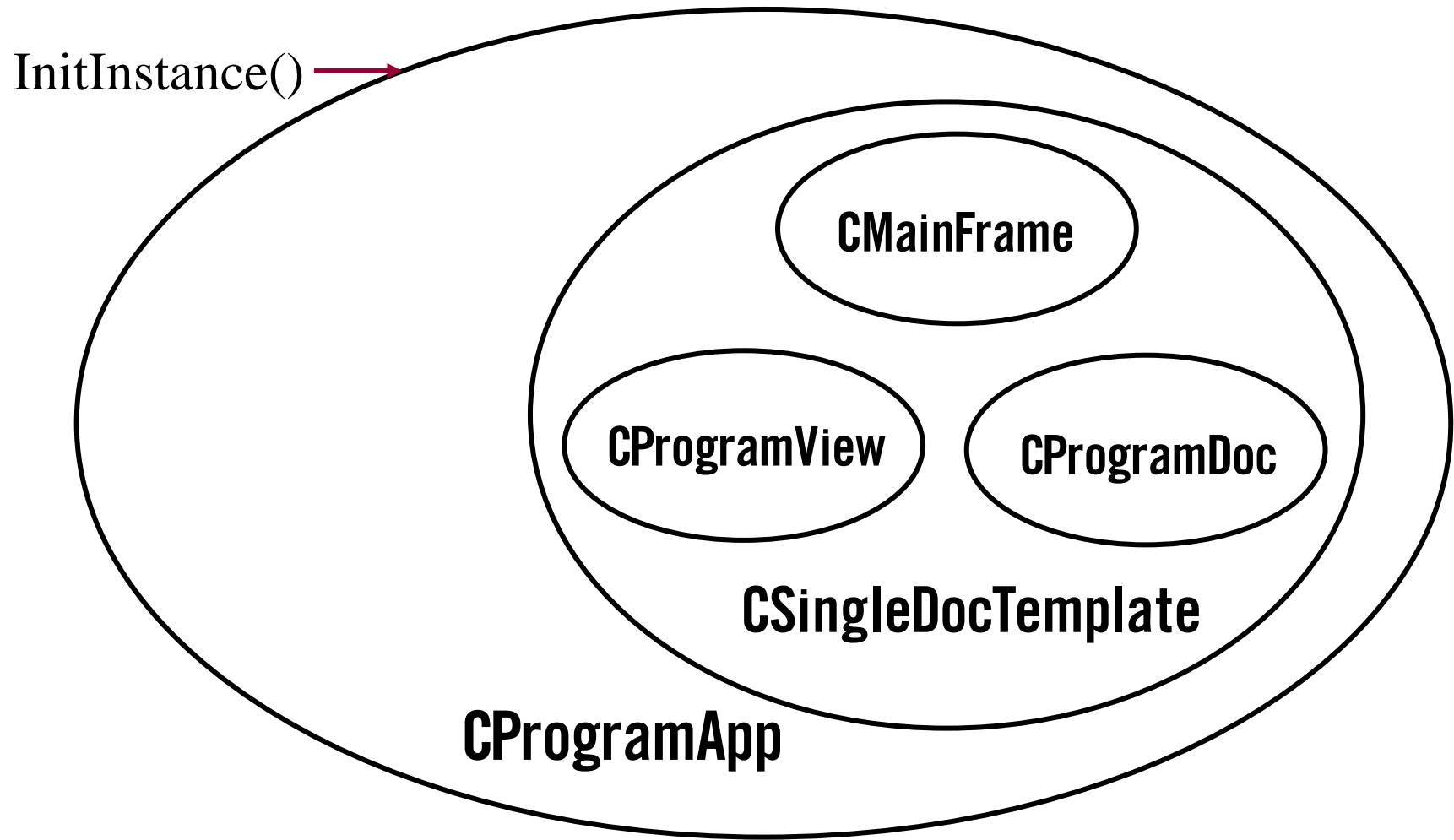
```
class Geometry : Object { // == class Geometry : public Object {  
public:  
    Geometry( );  
    ~Geometry( );  
    virtual void Draw( Window *pWnd ); // virtual method  
    BOOL IsDisplayed(void);  
    ....  
protected:  
    COLORREF          color;  
    ....  
private :  
    int      xPos, yPos;  
    double   xScale, yScale;  
    ...  
};  
class Point : Geometry {};  
class Line : public virtual Geometry { .... };  
class Polygon : protected Geometry {....};  
class Rectangle : private Geometry {....};  
....
```



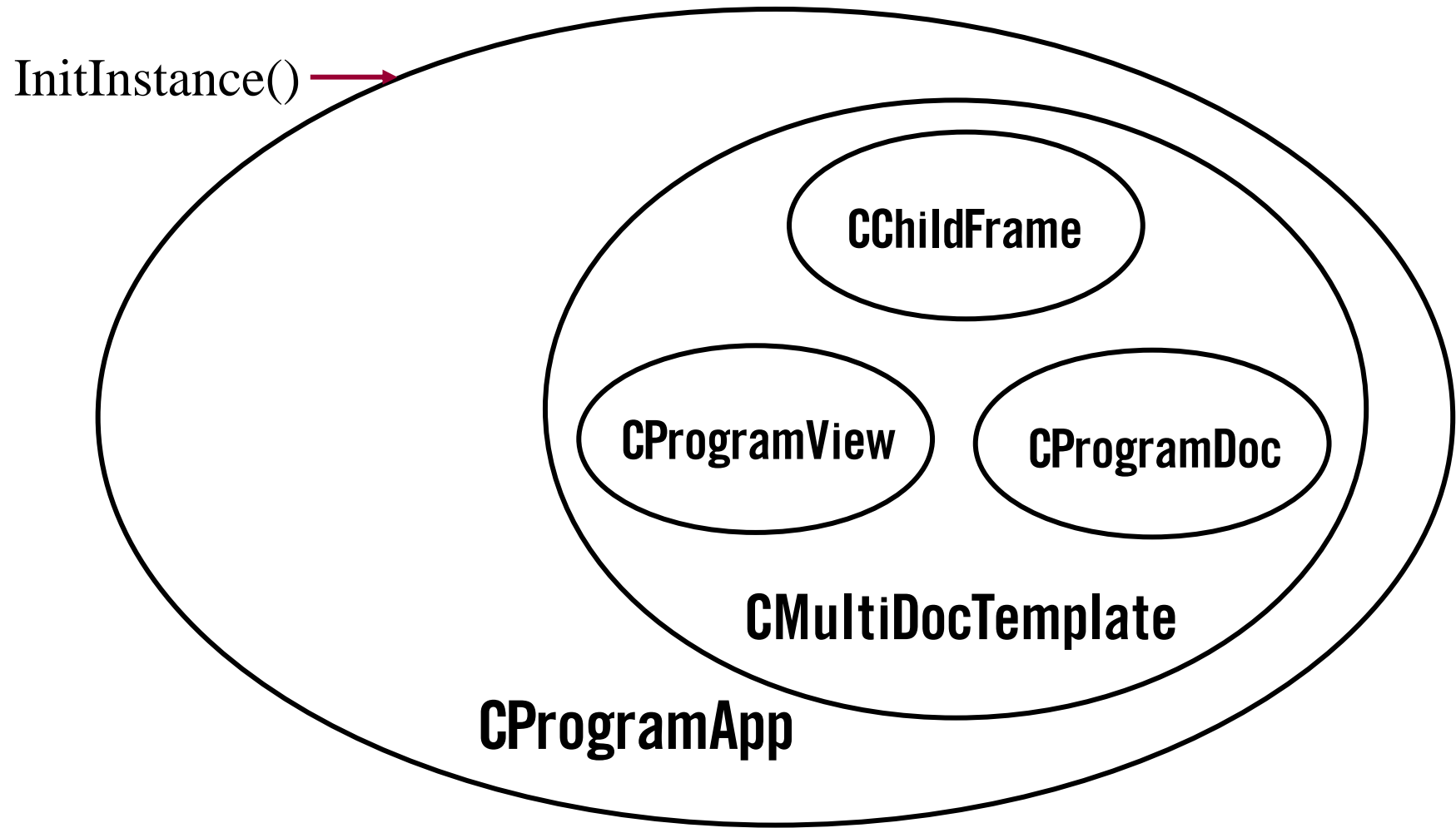
Cấu trúc 1 chương trình Dialog based đơn giản



Cấu trúc 1 chương trình SDI đơn giản



Cấu trúc 1 chương trình MDI đơn giản



2.2 Ngôn ngữ Java

1. Hỗ trợ 'interface' (1 dạng của type) và class.
2. Hỗ trợ Đơn thừa kế.
3. Dùng 'abstract class' để định nghĩa interface.
4. Tầm vực truy xuất các thành phần.
5. Hỗ trợ package
6. Đa hình đầy đủ.
7. Chỉ hỗ trợ đối tượng tạm trong session JVM
8. Override function khi thừa kế.
9. Có thể định nghĩa function overloaded.



Hỗ trợ Class và Interface

1. Chủ yếu dùng class để định nghĩa kiểu cho các biến, thuộc tính.

Có thể dùng interface để định nghĩa kiểu cho các biến, thuộc tính. Đối tượng chỉ có thể chứa tham khảo đến đối tượng khác.

2. Phải gọi hàm tạo đối tượng 1 cách tường minh, nhưng không được xóa đối tượng.

```
interface T1 {...}
```

```
class C1 extends RootClass implements T1 {...}
```

```
T1 p1;
```

```
C1 o1; // o1 chứa tham khảo đến đối tượng C1
```

```
p1 = o1 = New C1;
```

3. Interface chỉ được dùng trong trường hợp đặc biệt và không tương đương với abstract type.

4. Đơn thừa kế trong định nghĩa class \Rightarrow mối quan hệ thừa kế giữa các class khá đơn giản.



Hỗ trợ abstract class

5. Hỗ trợ khái niệm "abstract class" để định nghĩa class chứa thông tin interface và không cho phép 'instanciate' đối tượng. Bạn chỉ có thể dùng class 'abstract class' để đặc tả kiểu cho các biến hoặc định nghĩa các class con.

```
class abstract Geometry {           // abstract class  
protected int           xPos, yPos;  
protected double        xScale, yScale;  
protected COLORREF color;  
...  
public abstract Draw(Graphics g); // abstract function  
...  
};
```

Abstract class có thể chứa đầy đủ các hiện thực bên trong, nhưng thường chỉ có chứa các 'abstract function'.

Tầm vực truy xuất các thành phần

6. Tầm vực truy xuất các thành phần trong đối tượng :

private : thành phần bị che dấu hoàn toàn.

protected : che dấu bên ngoài nhưng cho phép các đối tượng con, cháu, chắt... truy xuất.

public : cho phép tất cả mọi nơi truy xuất.

friendly : cho phép mọi phần tử trong package truy xuất. Đây là tầm vực default và không có từ khóa tầm vực tường minh.



Hỗ trợ package

7. Package là đơn vị quản lý tầm vực của java, có thể chứa nhiều class.

```
package graphics;
```

```
public class Circle extends Graphic implements Draggable {  
    . . .  
}
```

Tất cả mọi phần tử được định nghĩa trong 1 file source đều thuộc 1 package : tên được qui định bởi phát biểu package hay là package default.

Nhiều file source có thể thuộc cùng 1package (dùng cùng tên trong phát biểu package).



Hỗ trợ đầy đủ tính đa hình

8. Tất cả các public function được quản lý trong 1 danh sách "public function table".

địa chỉ function 1
địa chỉ function 2
địa chỉ function 3
địa chỉ function i
địa chỉ function n

Các đối tượng đều 'tạm thời'

9. Các đối tượng chỉ tồn tại tạm thời trong 1 session chạy JVM.
Bạn có thể tạo ra các đối tượng mới mà không cần xóa nó. Đối tượng sẽ tồn tại một khi còn tham khảo đến nó. Module Garbage Collection trong JVM sẽ chịu trách nhiệm phát hiện đối tượng 'rác' và xóa nó ra khỏi bộ nhớ JVM.
10. Có quyền 'override' bất kỳ function nào của class cha.
11. Cho phép định nghĩa các hàm 'overloaded' : cùng tên nhưng 'signature' khác nhau.

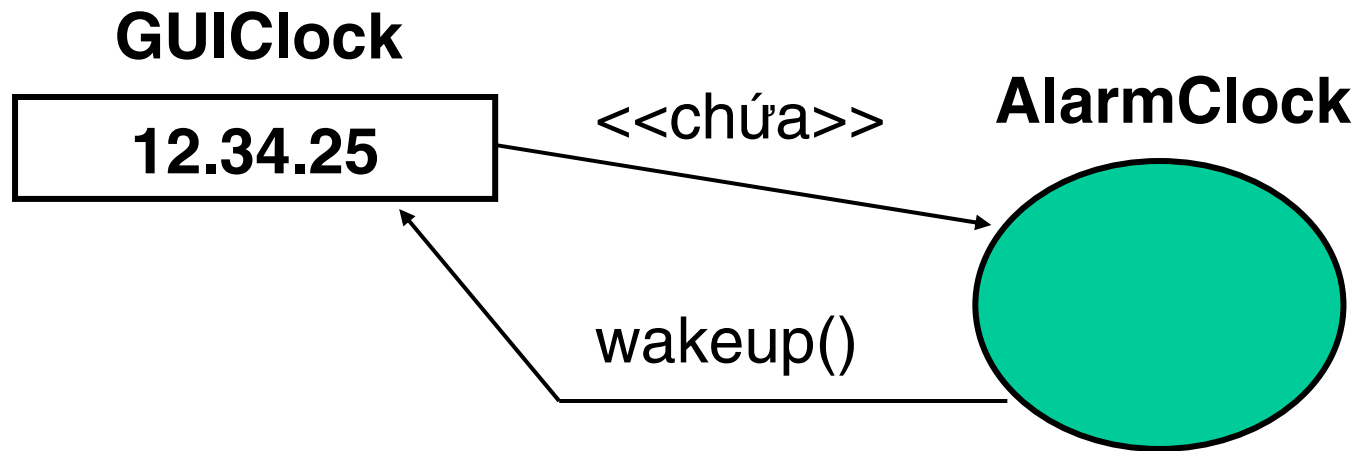


Thí dụ về chương trình Java

```
import java.net.*;
public class getnet {
    public static void main(String args[]) {
        try {
            if(args.length!=1) {
                System.out.println("Usage: java AddrLookupApp <HostName>");
                return;
            }
            InetAddress host = InetAddress.getByName(args[0]);
            String hostName = host.getHostName();
            System.out.println ("Host name : "+hostName);
            System.out.println ("IP address:"+host.getHostAddress());
        }
        catch (UnknownHostException e) {...}
    }
}
```



Thí dụ về chương trình Java



Thí dụ về các class Java

```
public class AlarmClock {  
    private static final int MAX_CAPACITY = 10;  
    private static final int UNUSED = -1;  
    private static final int NOROOM = -1;  
    private Sleeper[] sleepers = new Sleeper[MAX_CAPACITY];  
    private long[] sleepFor = new long[MAX_CAPACITY];  
    public AlarmClock () {  
        for (int i = 0; i < MAX_CAPACITY; i++)  
            sleepFor[i] = UNUSED;  
    }  
}
```



Thí dụ về các class Java

```
public synchronized boolean letMeSleepFor(Sleeper s, long time)
{
    int index = findNextSlot();
    if (index == NOROOM) {
        return false;
    } else {
        sleepers[index] = s;
        sleepFor[index] = time;
        new AlarmThread(index).start();
        return true;
    }
}
```



Thí dụ về các class Java

```
private synchronized int findNextSlot() {  
    for (int i = 0; i < MAX_CAPACITY; i++) {  
        if (sleepFor[i] == UNUSED)  
            return i;  
    }  
    return NOROOM;  
}  
  
private synchronized void wakeUpSleeper(int sleeperIndex) {  
    sleepers[sleeperIndex].wakeUp();  
    sleepers[sleeperIndex] = null;  
    sleepFor[sleeperIndex] = UNUSED;  
}
```



Thí dụ về các class Java

```
private class AlarmThread extends Thread {  
    int mySleeper;  
    AlarmThread(int sleeperIndex) {  
        super();  
        mySleeper = sleeperIndex;  
    }  
    public void run() {  
        try {  
            sleep(sleepFor[mySleeper]);  
        } catch (InterruptedException e) {}  
        wakeUpSleeper(mySleeper);  
    }  
}
```



Thí dụ về các class Java

```
public interface Sleeper {  
    public void wakeUp();  
    public long ONE_SECOND = 1000; // in milliseconds  
    public long ONE_MINUTE = 60000;    // in milliseconds  
}  
  
import java.applet.Applet;  
import java.awt.Graphics;  
import java.util.*;  
import java.text.DateFormat;  
  
public class GUIClock extends Applet implements Sleeper {  
    private AlarmClock clock;  
    public void init() {  
        clock = new AlarmClock();  
    }  
}
```



Thí dụ về các class Java

```
public void start() {  
    clock.letMeSleepFor(this, 1000);  
}  
public void paint(Graphics g) {  
    Calendar cal = Calendar.getInstance();  
    Date date = cal.getTime();  
    DateFormat dateFormatter =  
    DateFormat.getInstance();  
    g.drawString(dateFormatter.format(date), 5, 10);  
}  
public void wakeUp() {  
    repaint();  
    clock.letMeSleepFor(this, 1000);  
}
```



Vị trí & đời sống của các đối tượng C++

Trong VC++, đối tượng có thể nằm ở 3 không gian sau của phần mềm :

1. static data segment :

C1 c; //đời sống gắn liền với phần mềm

2. stack segment :

```
void func1() {
```

```
C1 d; //đời sống gắn liền với mỗi lần chạy function này
```

```
...
```

```
}
```

3. Heap (dynamic data segment) :

C1 *p = new C1(5,3.1); //đến lúc bị delete tường minh



Cơ chế kích hoạt constructor của đối tượng

Giả sử class C con của B, B con của A, A con của Root, mỗi class có thể có nhiều constructor “overloaded”. Thí dụ trong class C :

```
void C::C() {...}
```

```
void C::C(int i) {...}
```

```
void C::C(int i, double d): B(i): A(): Root(i,d) {...}
```

Xét :

```
C *p1 = new C(5);
```

Lệnh này sẽ kích hoạt các constructor chạy như sau : **Root()** → **A()** → **B()** → **C(5)**. Mặc định thì constructor không tham số của class cha sẽ được gọi.

```
C *p2 = new C(5, 3.14);
```

Lệnh này sẽ kích hoạt các constructor chạy như sau : **Root(5,3.14)** → **A()** → **B(5)** → **C(5,3.14)**. Constructor của class con qui định cụ thể từng constructor cha nào sẽ được chạy.

Cơ chế kích hoạt destructor của đối tượng

Giả sử class C con của B, B con của A, A con của Root, mỗi class có thể có 1 destructor và destructor thường không cần có tham số. Thí dụ trong class C :

```
void C::~~C() {...}
```

Xét :

```
C *p1 = new C(5);
```

```
delete (p1);
```

Lệnh này sẽ kích hoạt các destructor chạy như sau : $\sim C() \rightarrow \sim B() \rightarrow \sim A() \rightarrow \sim \text{Root}()$.

Cú pháp & ngữ nghĩa gọi hàm & gọi thông điệp

Trong thân 1 function cổ điển :

`C *pobj = new D(); // = @d;`

`D d; //class D là con của C`

`C obj = d;`

`func1(...); //::func1(...);` gọi hàm cổ điển func1

`C::func1(...); //C_func1(...);` gọi hàm func1 của class C

`obj.func1(...); //C_func1(...);` gọi hàm func1 của class C cho dù obj đang chứa đối tượng class D

`pobj->func1(...); //C_func1(...);` nếu func1 không phải virtual function thì sẽ gọi hàm func1 của class C (dù pobj đang pointer đến đối tượng của class nào đi chăng nữa)

`pobj->func1(...);` //đoạn code liên kết động để đảm bảo đa xạ, nếu func1 là hàm virtual.



Cú pháp & ngữ nghĩa gọi hàm & gọi thông điệp

Trong thân 1 tác vụ của 1 class (A) :

`C *pobj = new C();`

`D d; //class D là con của C`

`C obj = d;`

`::func1(...);` gọi hàm cổ điển func1

`C::func1(...); //C_func1(...);` gọi hàm func1 của class C

`obj.func1(...); //C_func1(...);` gọi hàm func1 của class C cho dù c đang chứa đối tượng class D

`pobj->func1(...); //C_func1(...);` nếu func1 không phải virtual function thì sẽ gọi hàm func1 của class C (dù pobj đang pointer đến đối tượng của class nào đi chăng nữa)

`pobj->func1(...);` //đoạn code liên kết động để đảm bảo đa xạ, nếu func1 là hàm virtual.



Cú pháp & ngữ nghĩa gọi hàm & gọi thông điệp

Trong thân 1 tác vụ của 1 class (A) :

`C *pobj = new C();`

`D d; //class D là con của C`

`C obj = d;`

`this->func1();` // hay lệnh kể

`func1(...);` //sẽ được xử lý theo 1 trong 3 nghĩa sau :

1. `::func1(..);` //gọi hàm cổ điển func1 nếu func1 không tồn tại trong class A. Nên viết theo qui ước : `::func1(...);`
2. `A_func1(...);` //nếu func1 là tác vụ thường trong class A (không phải virtual function) thì sẽ gọi hàm func1 của class A (dù pobj đang pointer đến đối tượng của class nào đi chăng nữa). Nên viết theo qui ước : `func1(...);`
3. //đoạn code liên kết động để đảm bảo đa xạ, nếu func1 là hàm virtual. Nên viết theo qui ước : `this->func1(...);`

Vị trí của các đối tượng Java

Trong Java, đối tượng chỉ có thể nằm ở không gian hệ thống (máy ảo JVM), trong phần mềm, ta chỉ có biến tham khảo đến đối tượng :

```
C p; //biến tham khảo đến đối tượng
```

```
p = new C(5,3.1); //nhờ hệ thống tạo đối tượng mới.
```

Trong Java, phần mềm không cần xóa đối tượng, chuyện này do hệ thống tự động làm thông qua module "dọn rác".



Cơ chế kích hoạt constructor của đối tượng

Giả sử class C con của B, B con của A, A con của Root, mỗi class có thể có nhiều constructor “overloaded”. Thí dụ trong class C :

```
void C() { super (1, 10.3);...}
```

```
void C(int i) {...}
```

```
void C(int i, double d) {  
    super(i); ...}
```

Xét :

```
C p1 = new C(5);
```

Lệnh này sẽ kích hoạt các constructor chạy như sau : **Root()** → **A()** → **B()** → **C(5)** theo cơ chế mặc định nếu các constructor không chứa lệnh super.

```
C p2 = new C(5, 3.14);
```

Lệnh này sẽ kích hoạt các constructor chạy như sau : **Root(5,3.14)** → **A()** → **B(5)** → **C(5,3.14)** theo yêu cầu của lệnh super trong từng constructor của các class (lệnh supper qui định cụ thể constructor cha nào sẽ được chạy).



Cơ chế kích hoạt destructor của đối tượng

Giả sử class C con của B, B con của A, A con của Root, mỗi class có thể có 1 destructor (có tên là finalize). Destructor thường không cần có tham số. Thí dụ trong class C :

```
void finalize() {...}
```

Xét :

```
C p1 = new C(5);
```

```
p1 = new C(1);
```

Lệnh này sẽ kích hoạt các destructor chạy như sau : finalize của C → finalize của B → finalize của A → finalize của Root.



Cú pháp & ngữ nghĩa gọi hàm & gọi thông điệp

Trong thân 1 tác vụ của 1 class (A) :

`C pObj = new D();` //class D là con của C

`C.func1(...);` // `C_func1(...)`; gọi hàm `func1` của class C

`this.func1();` // hay lệnh kể

`pObj.func1(...);` //đoạn code liên kết động để đảm bảo đa xạ,

`func1(...);` //sẽ được xử lý theo 1 trong 2 nghĩa sau :

1. `A.func1(..);` //nếu `func1` là tác vụ private trong class A. Nên viết theo qui ước : `func1(...);`
2. //đoạn code liên kết động để đảm bảo đa xạ, nếu `func1` không là hàm private. Nên viết theo qui ước : `this.func1(...);`











Chương 3

NGUYÊN TẮC DỊCH OOP

- ➡ Dịch abstract type
- ➡ Dịch class

Tổng quát về vấn đề dịch OOP

- Chương trình là 1 tập các đối tượng sống và tương tác lẫn nhau.
- Các đối tượng thuộc 1 số loại nhất định (n)
- Mỗi loại đối tượng được miêu tả bởi 1 type + 1 class
- Chương trình là tập n định nghĩa type + class
- Dịch chương trình OOP là vòng lặp dịch n type + n class.
- Ta sẽ miêu tả qui trình dịch 1 type và 1 class



Dịch 1 abstract type

- Abstract type chỉ chứa thông tin trừu tượng (interface), không miêu tả sự hiện thực → Kết quả việc dịch 1 type chỉ dừng lại cây ngữ nghĩa của type tương ứng để phục vụ việc kiểm tra kiểu, chứ không tạo code mã máy.
- Chỉ cần 3 bước : duyệt từ vựng, phân tích cú pháp, phân tích ngữ cảnh.
- Nên dùng công cụ hỗ trợ như LEX, YACC.

Dịch 1 class

- Dịch class là công việc chính của chương trình dịch OOP.
- Gồm 2 công việc chính : dịch thuộc tính dữ liệu và dịch các method (hay các internal function).
- Cần đầy đủ các bước : duyệt từ vựng, phân tích cú pháp, phân tích ngữ cảnh, tạo mã.
- Nên dùng công cụ hỗ trợ như LEX, YACC.



Dịch thuộc tính dữ liệu

- class → cấu trúc record

```
class C1 : C0 {  
    double d;  
    int i ;  
    ...  
public :  
    int proc4(int i);  
    void proc5 (double d);  
    ...  
};
```

typedef struct {

// import các field từ cấu trúc
// được sinh ra từ C0

// các field tương ứng với C1

double C1_d;

int C1_i;

...

// các field dữ liệu điều khiển

// tự tạo bởi chương trình dịch

void (*pvfaddr)() ;

} C1;

Dịch thuộc tính dữ liệu (tt)

- mỗi class → 1 record cố điển.
- tên class → tên record.
- copy các field dữ liệu của cấu trúc sinh ra từ class cha.
- chuyển từng thuộc tính của class thành từng field của record, 'tuyệt đối hóa' tên của thuộc tính để tránh nhập nhằng.
- thêm các field dữ liệu điều khiển phục vụ cho run-time : thí dụ bảng địa chỉ các method của đối tượng (pvftbl).



Dịch thuộc tính dữ liệu (tt)

- cấu trúc record được dịch ra mã máy thành 1 vùng nhớ liên tục có độ dài bằng độ dài của record.
 - field C1 o1; C1_o1 db dup (sizeof(C1))
- truy xuất 1 thuộc tính dữ liệu trở thành việc truy xuất ô nhớ dùng cách định địa chỉ chỉ số :
 - o1.i = 5; mov bx, C1_o1
 mov [bx+8], 5

Tạo bảng địa chỉ các method

```
class C1 : C0 {  
    int i ;  
    double d;  
    ...  
public :  
    v void proc2() {...};  
    v int proc4(int i, double k);  
    v void proc5 (double d);  
    void proc6 (double d);...  
};
```

pvftbl

fname

faddr

0	"proc1"	C0_proc1
1	"proc2"	C1_proc2
2	"proc3"	C0_proc3
3	"proc4"	C1_proc4
4	"proc5"	C1_proc5
5

Tạo bảng địa chỉ các method (tt)

- tạo bảng địa chỉ gồm C1METHCNT phần tử (C1METHCNT là số method của class hiện hành, kể cả các method thừa kế).
- mỗi phần tử được nhận dạng qua chỉ số và gồm 2 thông tin chính : tên gọi nhớ của method và địa chỉ của method.
- copy bảng địa chỉ của class cha đã có.
- hiệu chỉnh lại các địa chỉ của các method bị override.
- thêm vào các method mới định nghĩa trong class hiện hành.



Dịch 1 method

```
int C1::proc4(int i,double k) {  
    C2 o2;  
    C2 *p2;  
    this->i = i;  
    d = k;  
    proc6(d);  
    o2.proc2(i,d);  
    p2 = New(C2);  
    p2->proc2(i,d);  
    ....  
};
```

```
int C1_proc4(C1* p, int i, double d) {  
    C2 o2; C2 *p2;  
    // truy xuất thuộc tính  
    p->C1_i = i; p->C1_d = d;  
    // gọi hàm  
    C1_proc6(p,p->C1_d);  
    C2_proc2(&o2, i,p->C1_d);  
    // gửi thông báo : kiểm tra, load,  
    // ánh xạ bảng địa chỉ method  
    for (i = 0; i < C2METHCNT; i ++)  
        if (strcmp ("proc2", p2->  
                    pvftbl[i].fname)==0) break;  
    (*pvftbl[i].faddr)(p2,i,p->C1_d);  
};
```

① // gửi thông báo : kiểm tra, load,
// ánh xạ bảng địa chỉ method
② for (i = 0; i < C2METHCNT; i ++)
if (strcmp ("proc2", p2->
pvftbl[i].fname)==0) break;
③ (*pvftbl[i].faddr)(p2,i,p->C1_d);

Dịch 1 method (tt)

- tên method được chuyển từ dạng 'tương đối' sang 'tuyệt đối' (nối kết tên class vào).
- thêm tham số đầu tiên cho hàm sinh ra : miêu tả tham khảo đến đối tượng mà hàm sẽ truy xuất các thuộc tính dữ liệu.
- tên thuộc tính được chuyển từ dạng 'tương đối' sang 'tuyệt đối' (nối kết tên class vào).
- gọi hàm internal → gọi hàm nhưng thêm tham số đầu tiên.
- gọi thông báo 3 bước :
 - kiểm tra, tìm, load và ánh xạ bảng địa chỉ các method của đối tượng.
 - tìm chỉ số của method cần gọi trong bảng (i).
 - gọi gián tiếp method thông qua địa chỉ phần tử thứ i trong bảng.

Tối ưu hóa code tạo ra

- có 2 vấn đề lớn trong quá trình dịch 1 class sang ngôn ngữ cổ điển.
 - bảng địa chỉ method chiếm nhiều chỗ.
 - tốn thời gian để phục vụ lệnh gọi thông báo : kiểm tra, load và ánh xạ bảng địa chỉ, tìm chỉ số method cần gọi và gọi gián tiếp qua địa chỉ trong bảng.
- 1 số chương trình dịch tìm cách tối ưu hóa các vấn đề này.
- slide sau là các tối ưu hóa của chương trình dịch C++ và cái giá phải trả.



Tối ưu hóa code tạo ra (tt)

- trong C++, tất cả đối tượng đều tạm thời và gắn chặt vào ứng dụng → bảng địa chỉ các method của các đối tượng luôn nằm sẵn trong không gian của ứng dụng.
- mỗi lần tạo đối tượng, biến pvftbl trong đối tượng được gán ngay địa chỉ đầu bảng method → không cần làm bước 1 cho mỗi lần gọi thông báo.
- C++ chỉ dùng mối quan hệ con/cha trong kiểm tra kiểu → công việc 2 được làm tại thời điểm dịch thay vì tại thời điểm gọi thông báo trong lúc chạy.
- cột tên gọi nhớ method không cần phải lưu trữ trong bảng địa chỉ các method.
- chỉ có các virtual function mới được giải quyết theo cơ chế đa hình, còn các function khác được dịch ra lời gọi trực tiếp.

Tối ưu hóa code tạo ra (tt)

- cái giá phải trả của việc tối ưu hóa trong C++ :
 - người lập trình phải tự quyết định method nào cần xử lý theo cơ chế đa hình, hàm nào không ? Nếu sự quyết định này sai thì sẽ gây lỗi khi chạy, mà là người thì khó lòng quyết định chính xác.
 - tính đa hình chỉ đúng giữa các đối tượng có mối quan hệ con/cha, ở đó thứ tự các địa chỉ method của mọi class con trong bảng địa chỉ luôn giống thứ tự các method tương ứng của class cha, tuy nhiên giữa 2 class bất kỳ thì không thể đảm bảo → kiểm tra kiểu trong C++ không thể nâng cấp lên bằng cách dùng mối quan hệ "conformity".



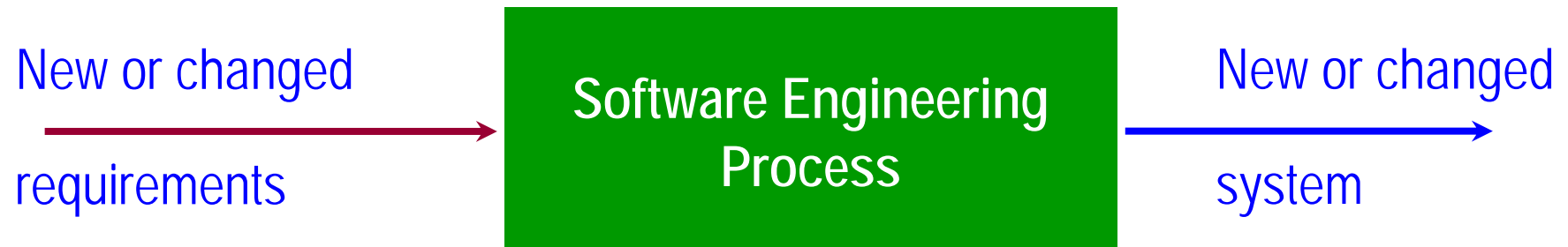
Chương 4

QUI TRÌNH HỢP NHẤT & UML

- ☞ Qui trình phát triển phần mềm hợp nhất
- ☞ Tổng quát về ngôn ngữ mô hình UML

What Is a Process?

- Defines Who is doing What, When to do it, and How to reach a certain goal.



Key concepts

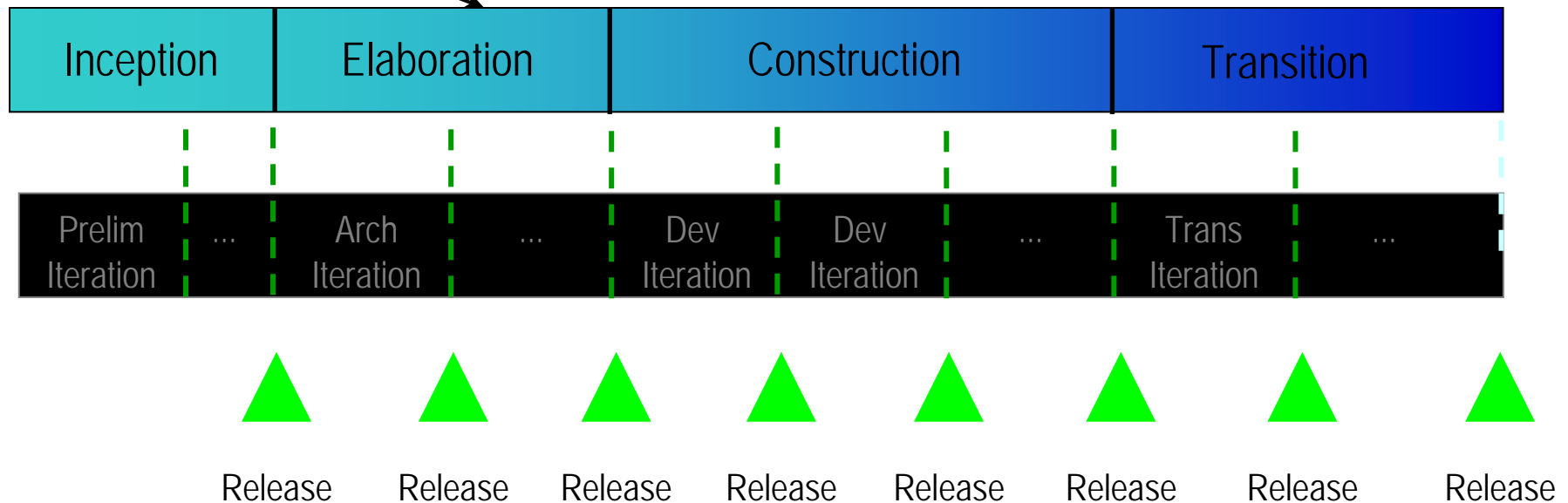
- Cycle → When does product happen?
- Phase, Iterations → When does architecture happen?
- Process Workflows → What does happen?
 - Activity, steps
- Artifacts → What is produced?
 - models
 - reports, documents
- Worker: Architect → Who does it?

Key concepts

time



Phase



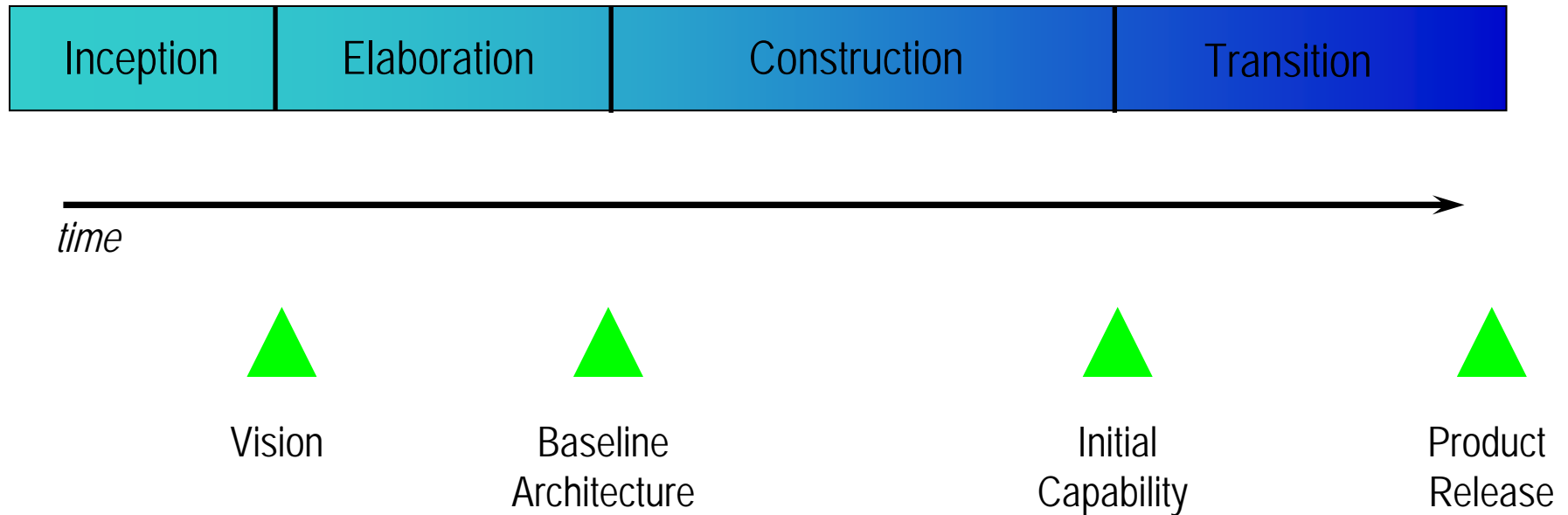
Lifecycle Phases



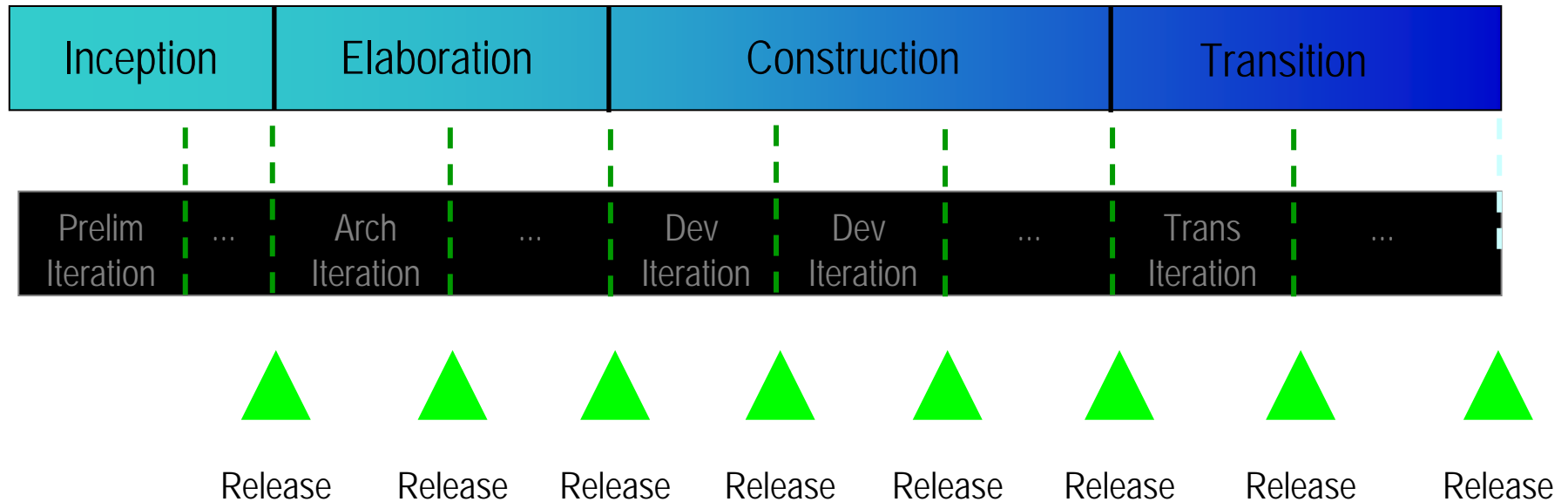
time →

- Inception Define the scope of the project and develop business case
- Elaboration Plan project, specify features, and baseline the architecture
- Construction Build the product
- Transition Transition the product to its users

Major Milestones



Phases and Iterations



An **iteration** is a sequence of activities with an established plan and evaluation criteria, resulting in an executable release

Iterations and Workflow

Phases

Core Workflows

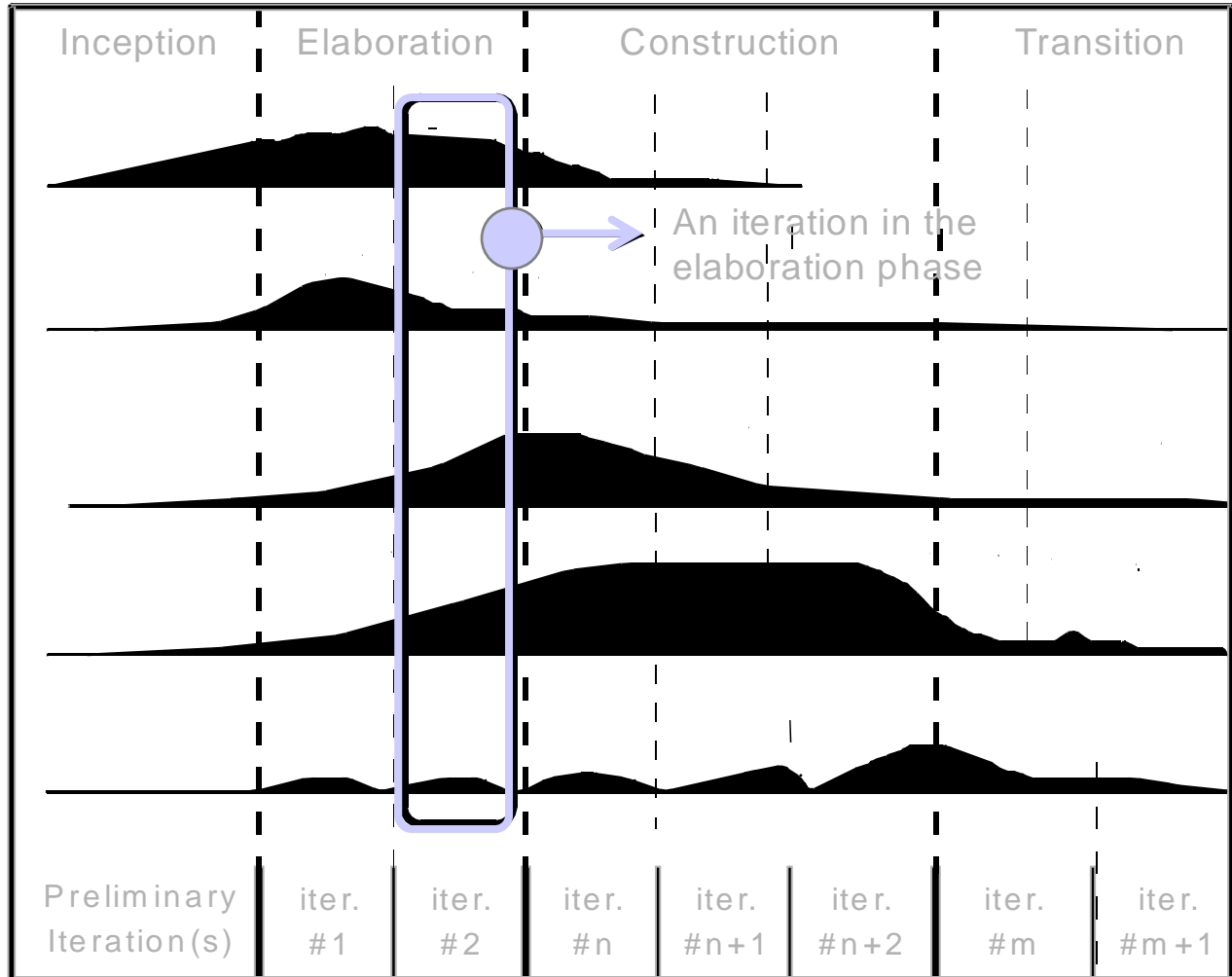
Requirements

Analysis

Design

Implementation

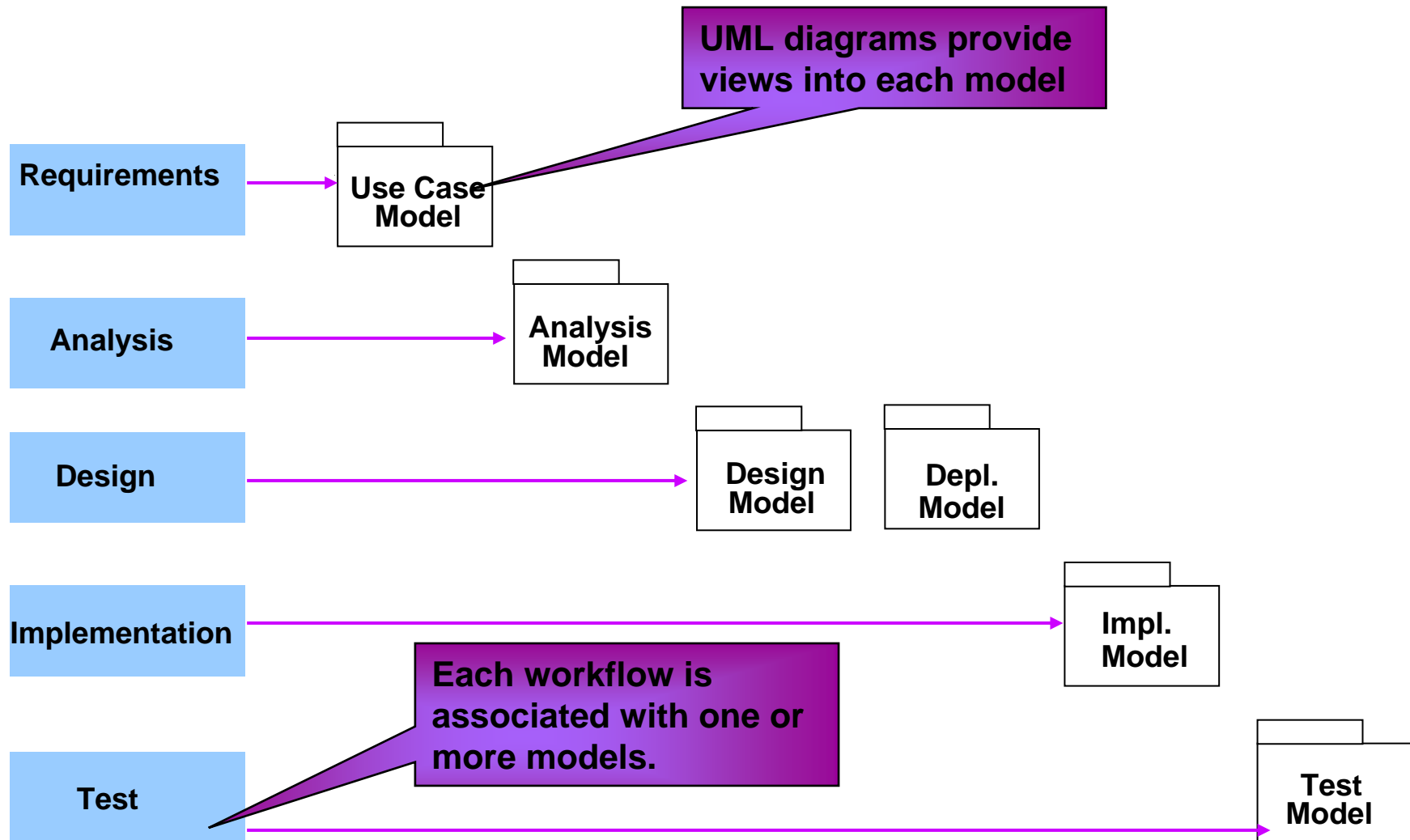
Test



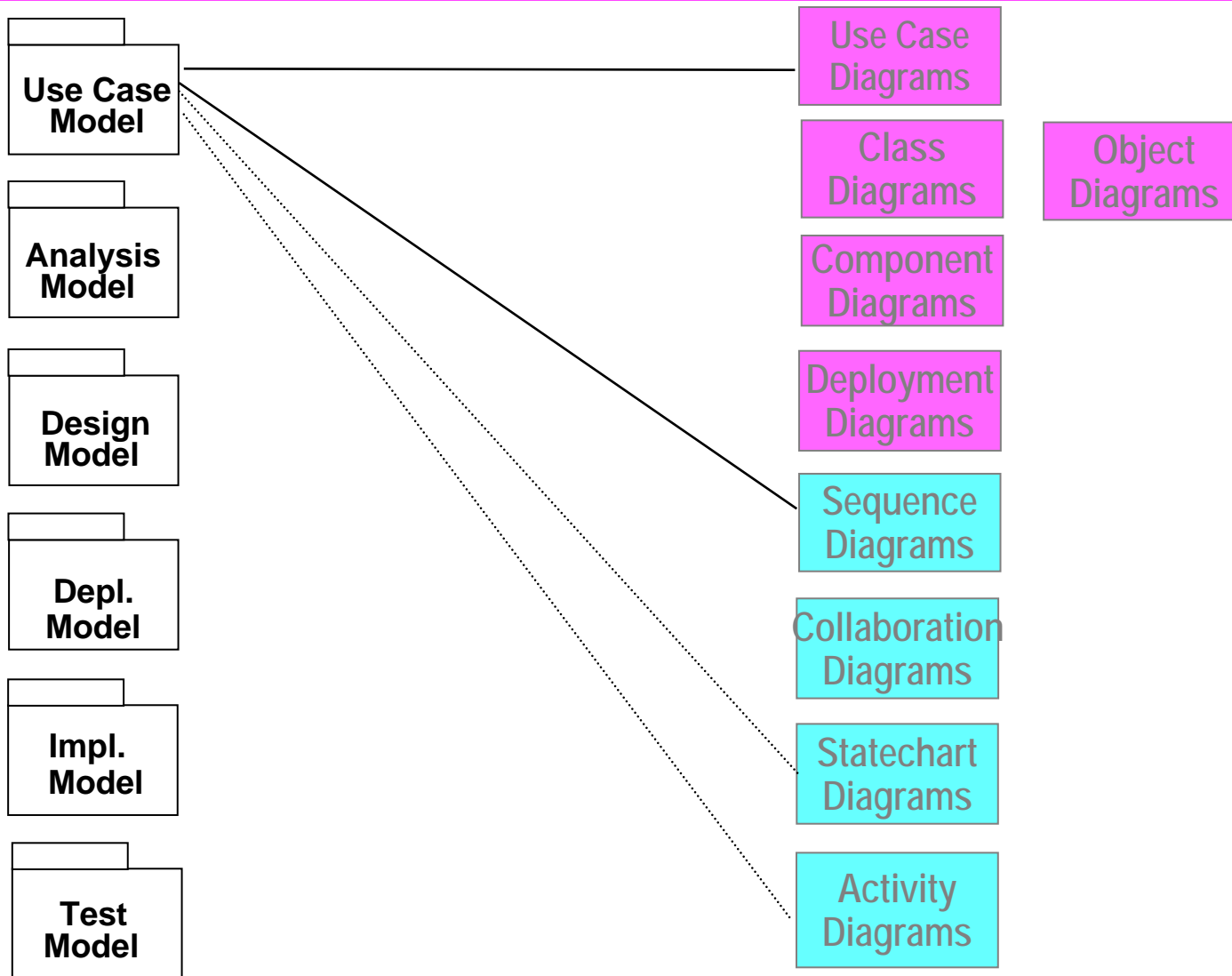
Iterations



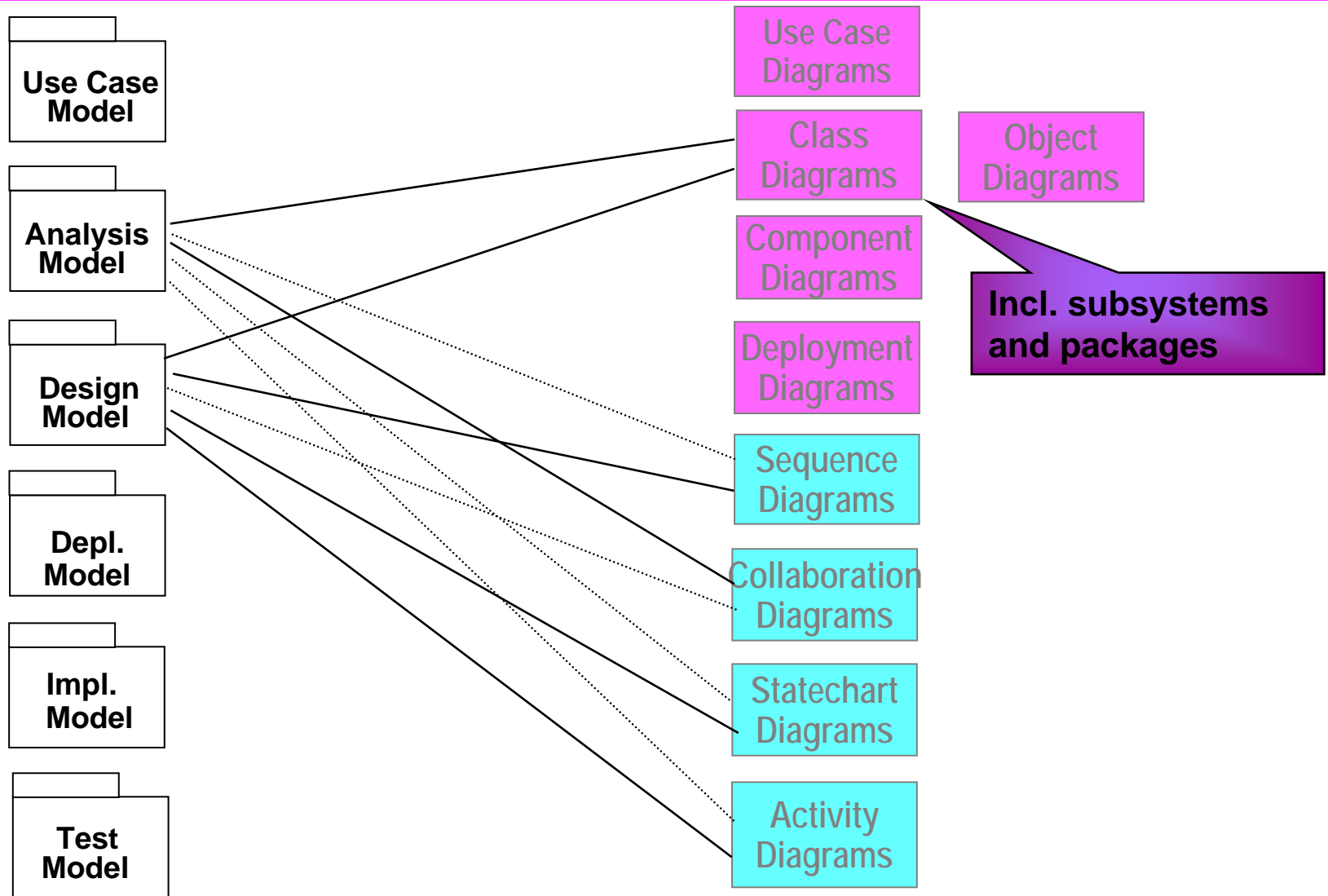
Workflows and Models



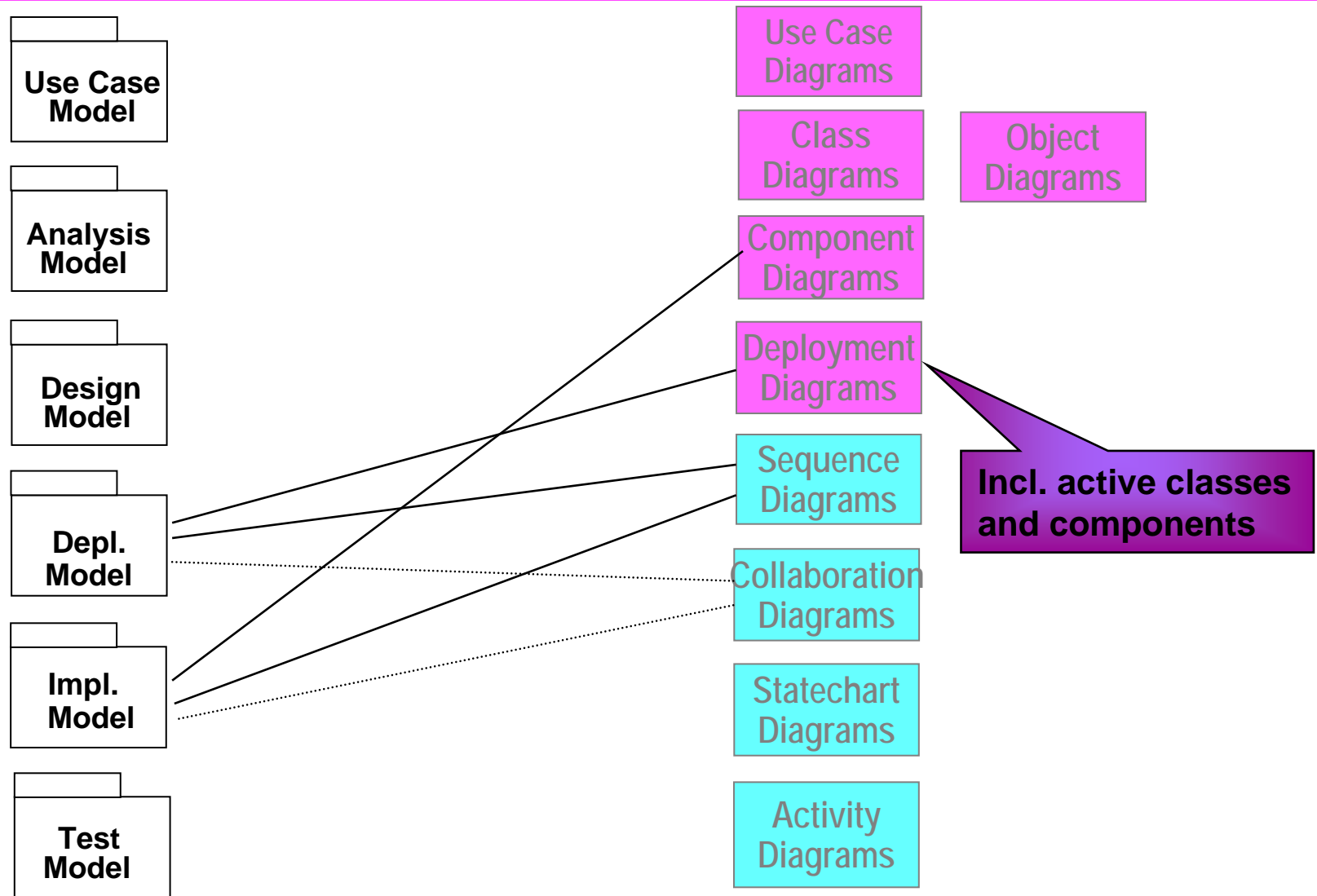
Use Case Model



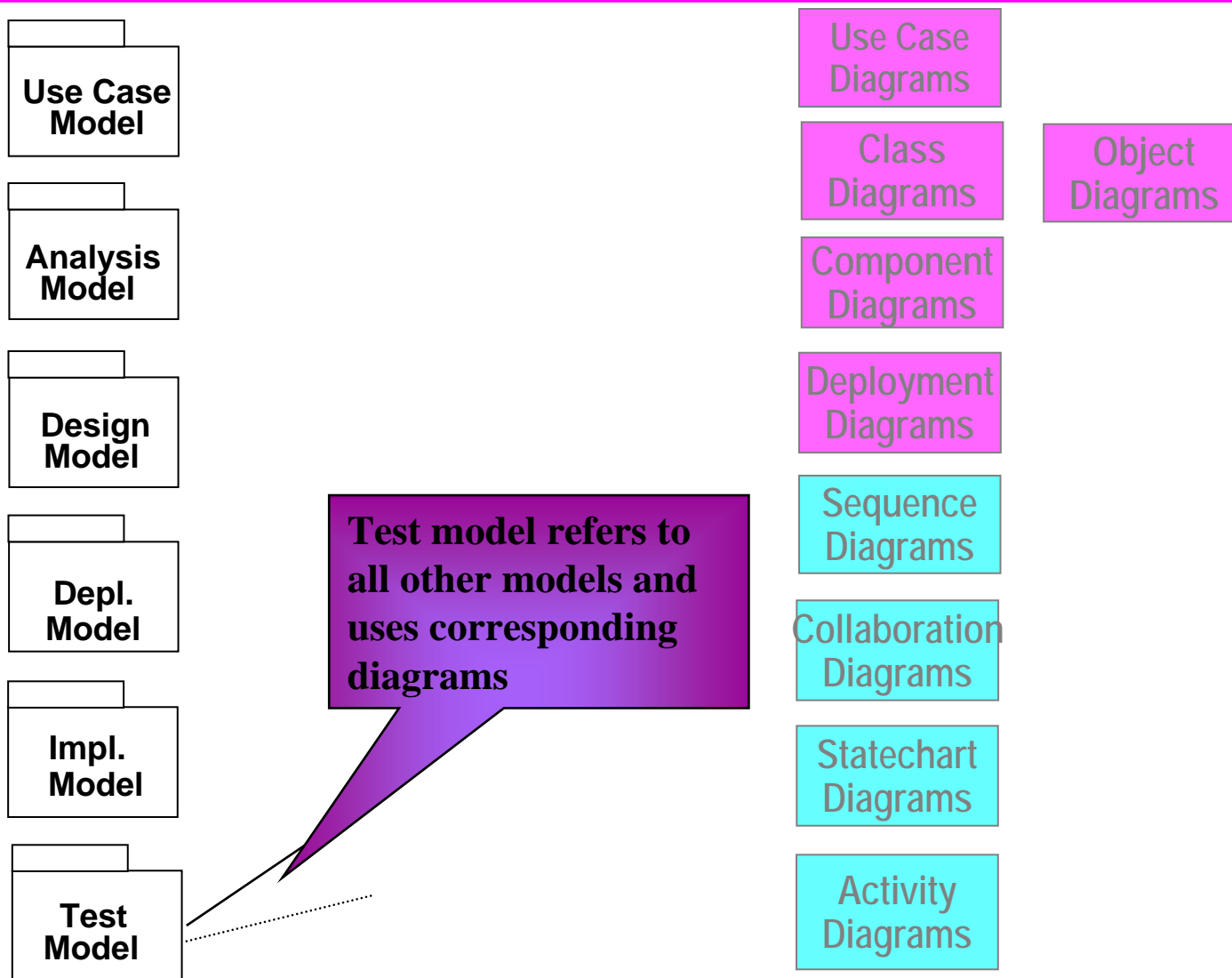
Analysis & Design Model



Deployment and Implementation Model



Test Model

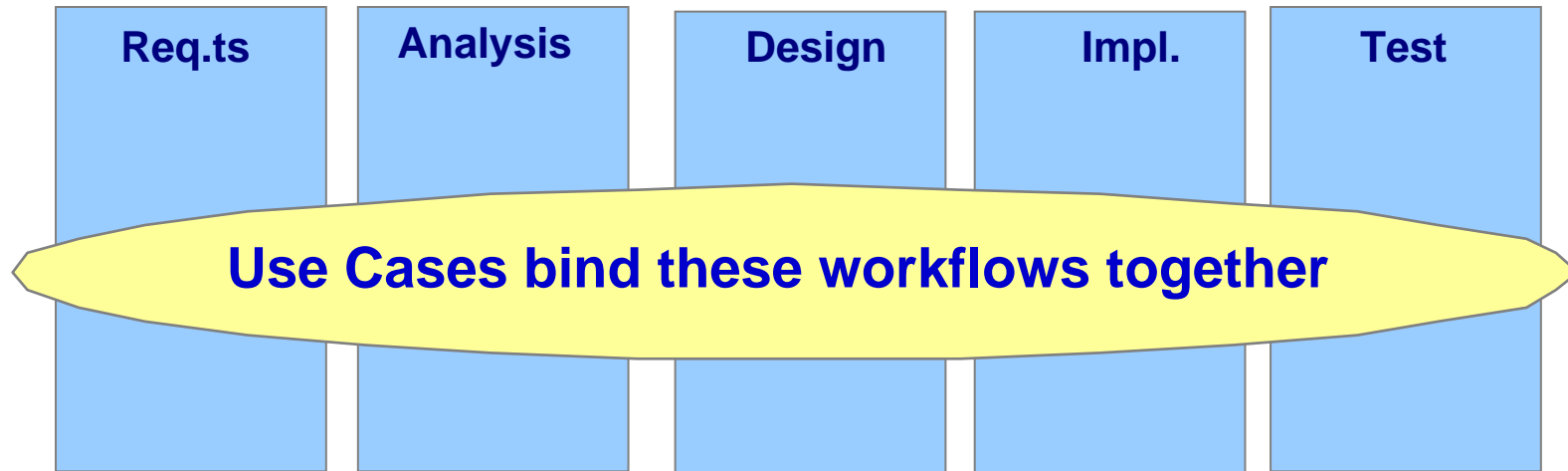


Overview of the Unified Process

- **The Unified Process is**
 - Iterative and incremental
 - Use case driven
 - Architecture-centric
 - Risk confronting



Use Case Driven



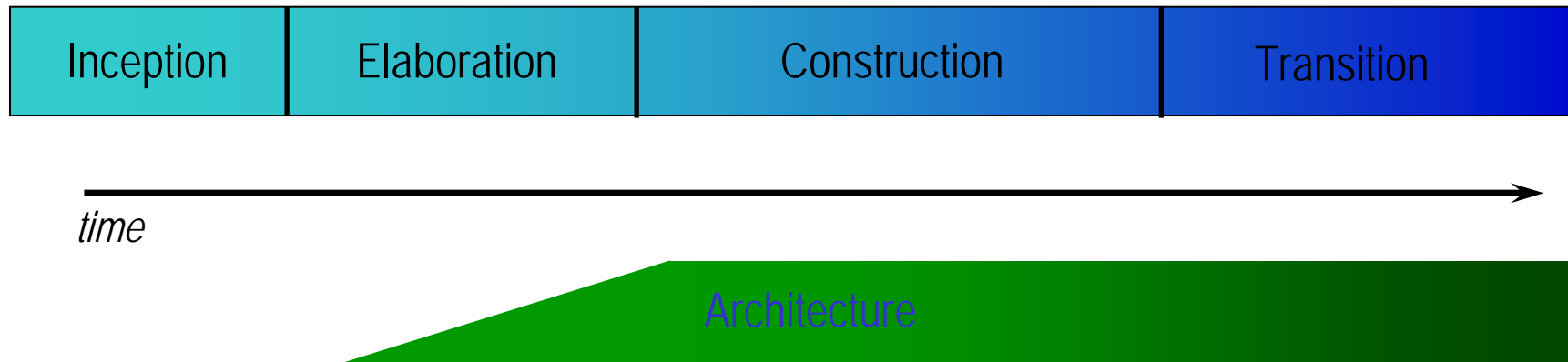
Use Cases Drive Iterations

- Drive a number of development activities
 - Creation and validation of the system's architecture
 - Definition of test cases and procedures
 - Planning of iterations
 - Creation of user documentation
 - Deployment of system
- Synchronize the content of different models



Architecture-Centric

- Models are vehicles for visualizing, specifying, constructing, and documenting architecture
- The Unified Process prescribes the successive refinement of an executable architecture



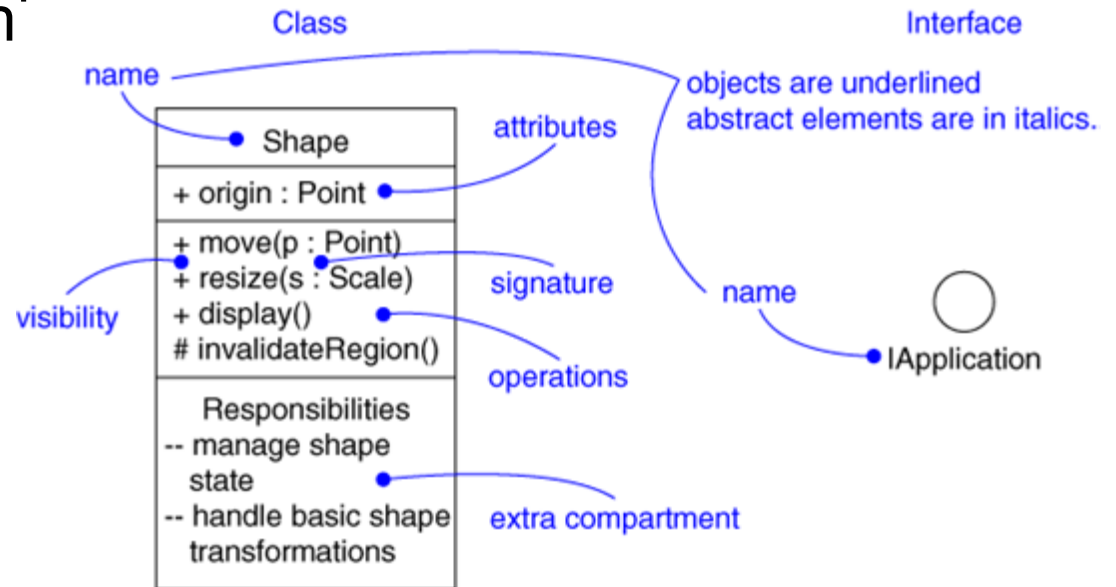
Overview of the UML

- The UML is a language for
 - visualizing
 - specifying
 - constructing
 - documentingthe artifacts of a software-intensive system
- There are 4 key elements :
 - Modeling elements
 - Relationships
 - Extensibility Mechanisms
 - Diagrams



Modeling Elements

- Structural elements
 - class, interface, collaboration, use case, active class, component, node
- Behavioral elements
 - interaction, state machine
- Grouping elements
 - package, subsystem
- Other elements
 - note



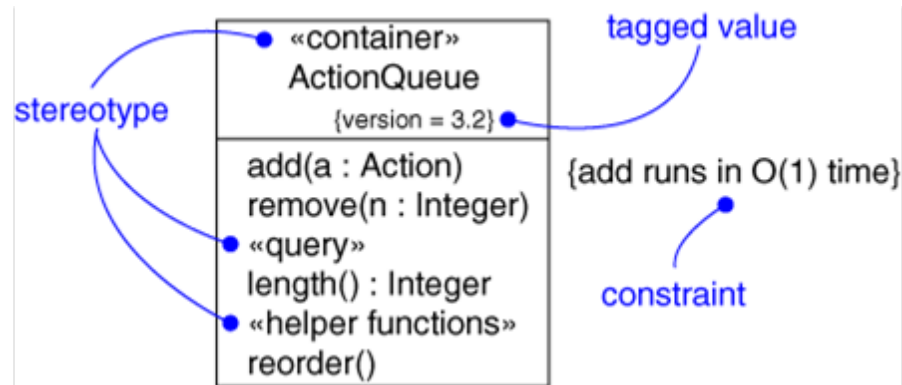
Relationships

- Dependency
- Association
- Generalization
- Realization



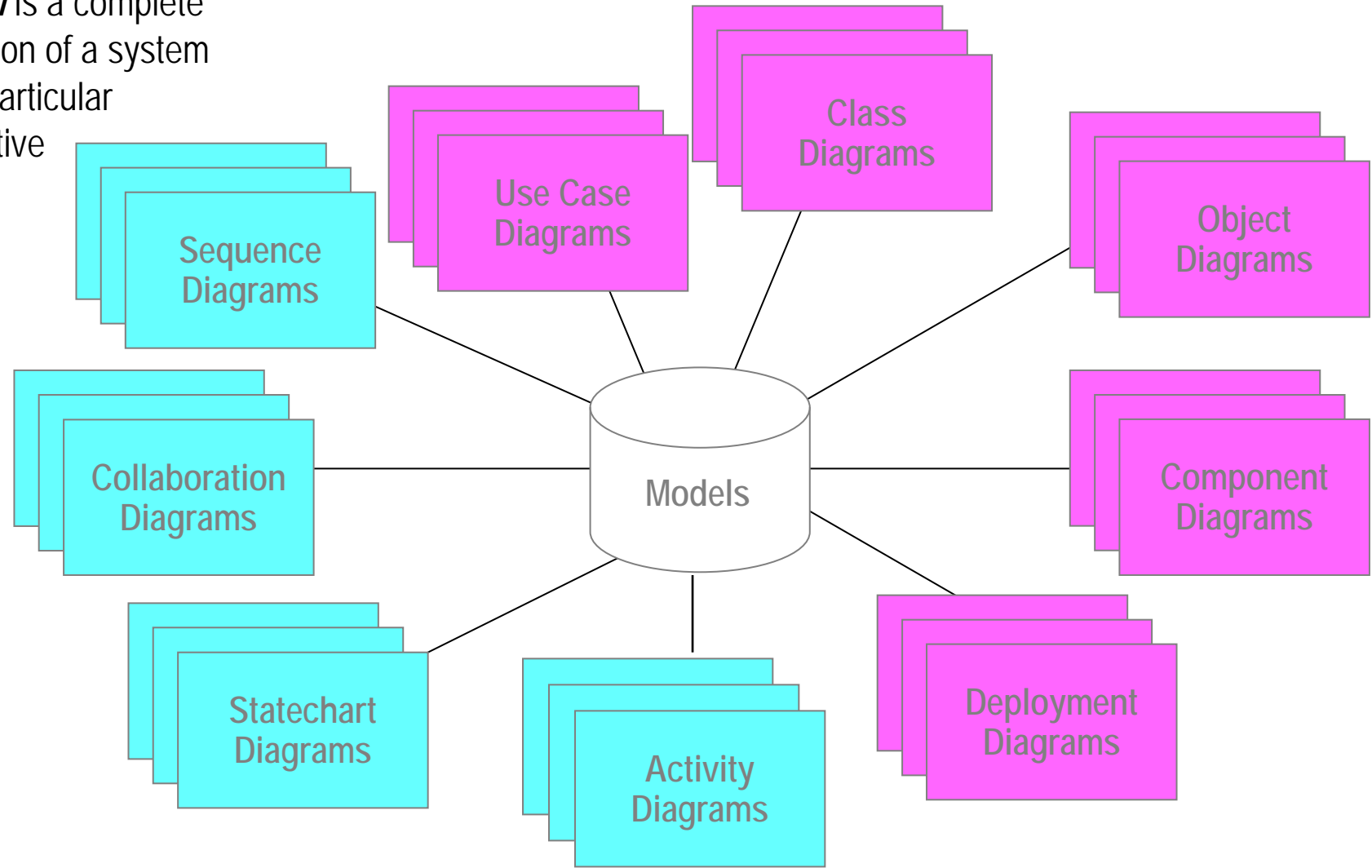
Extensibility Mechanisms

- Stereotype
- Tagged value
- Constraint



Models, Views, and Diagrams

A *model* is a complete description of a system from a particular perspective



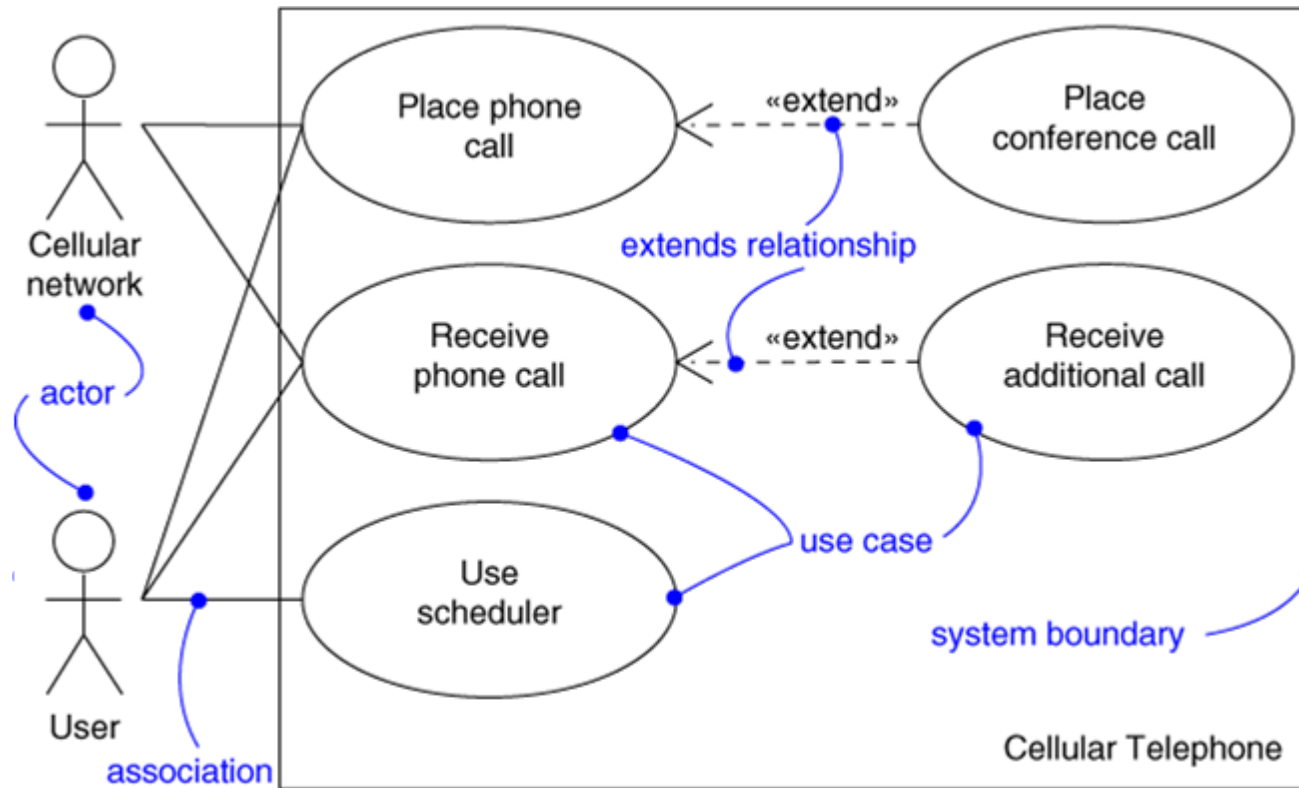
Diagrams

- A diagram is a view into a model
 - Presented from the aspect of a particular stakeholder
 - Provides a partial representation of the system
 - Is semantically consistent with other views
- In the UML, there are nine standard diagrams
 - Static views: use case, class, object, component, deployment
 - Dynamic views: sequence, collaboration, statechart, activity



Use Case Diagram

- Captures system functionality as seen by users



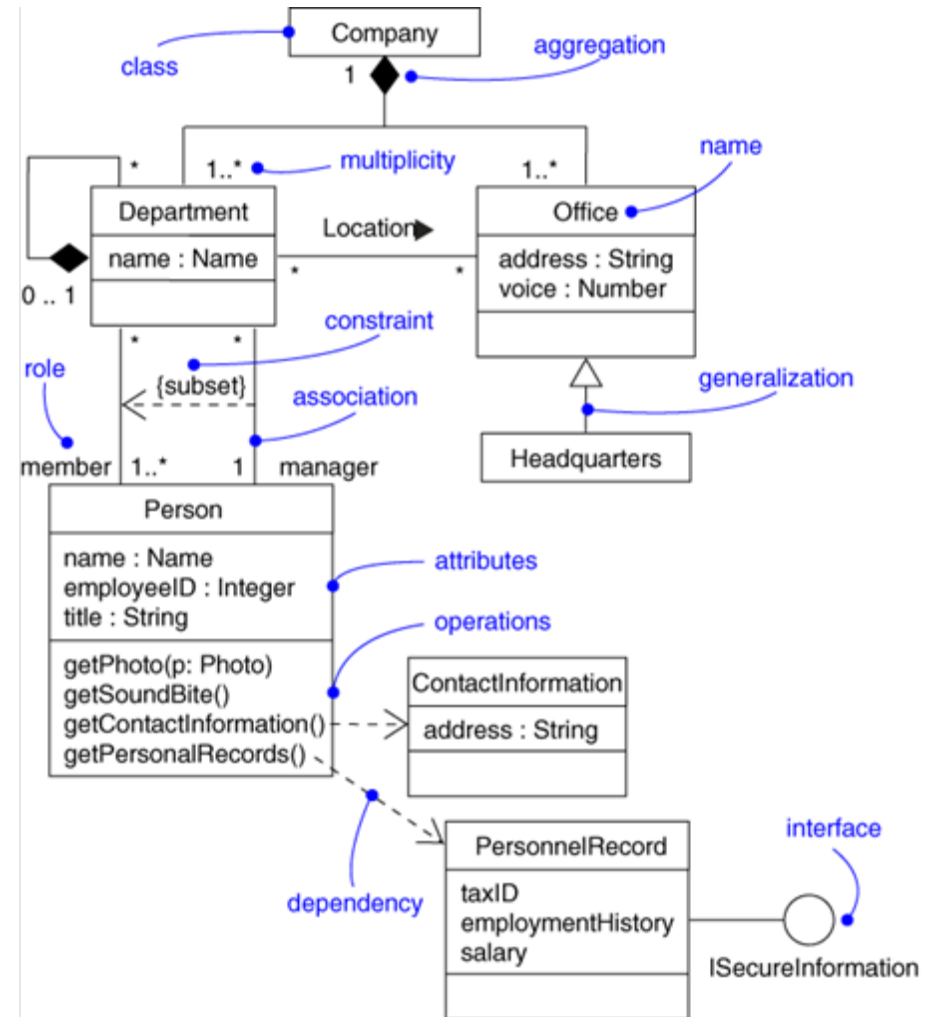
Use Case Diagram

- Captures system functionality as seen by users
- Built in early stages of development
- Purpose
 - Specify the context of a system
 - Capture the requirements of a system
 - Validate a system's architecture
 - Drive implementation and generate test cases
- Developed by analysts and domain experts



Class Diagram

- Captures the vocabulary of a system



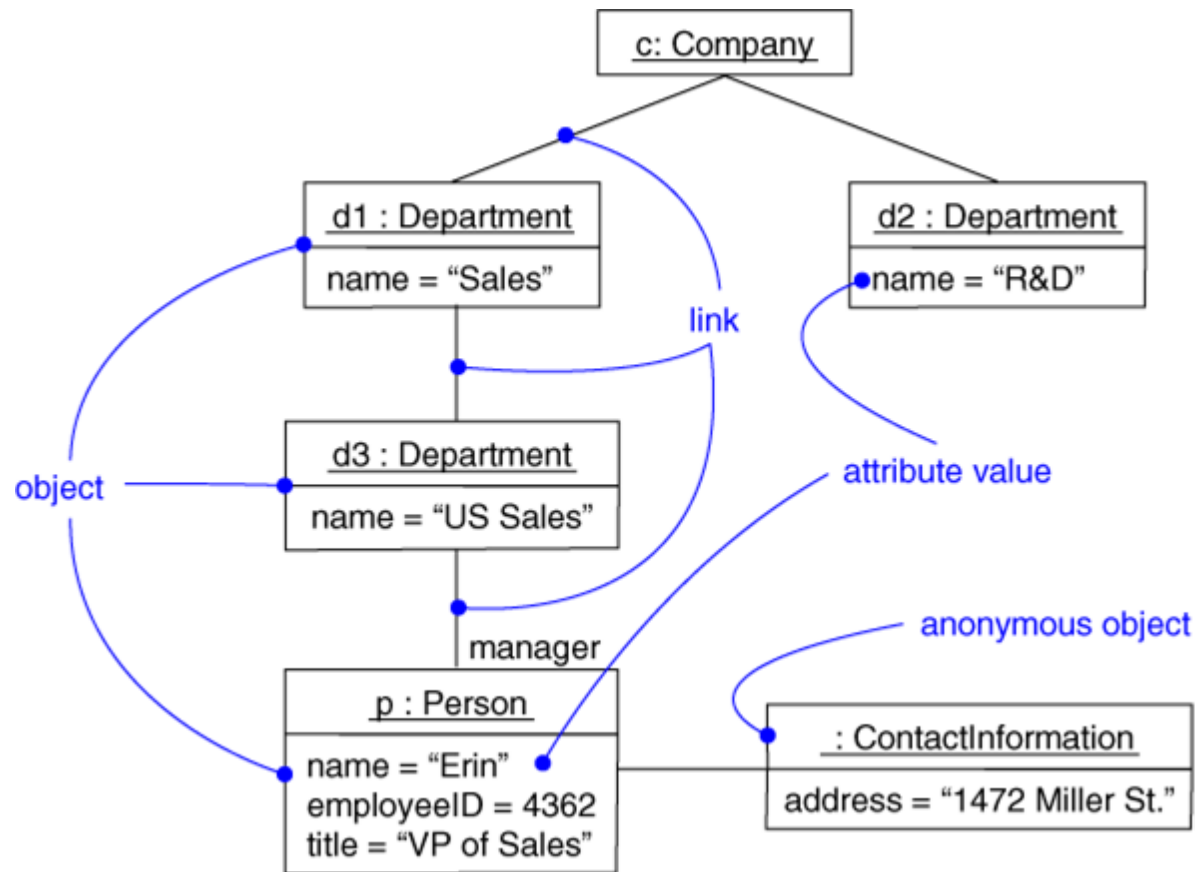
Class Diagram

- Captures the vocabulary of a system
- Built and refined throughout development
- Purpose
 - Name and model concepts in the system
 - Specify collaborations
 - Specify logical database schemas
- Developed by analysts, designers, and implementers



Object Diagram

- Captures instances and links



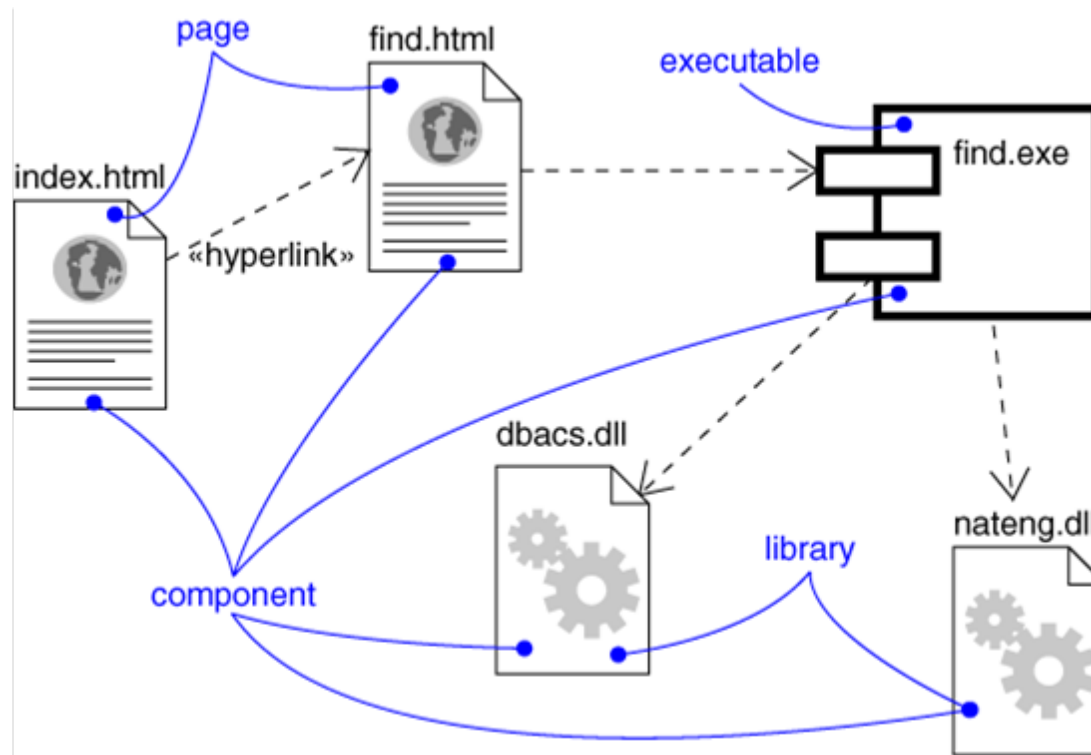
Object Diagram

- Shows instances and links
- Built during analysis and design
- Purpose
 - Illustrate data/object structures
 - Specify snapshots
- Developed by analysts, designers, and implementers



Component Diagram

- Captures the physical structure of the implementation



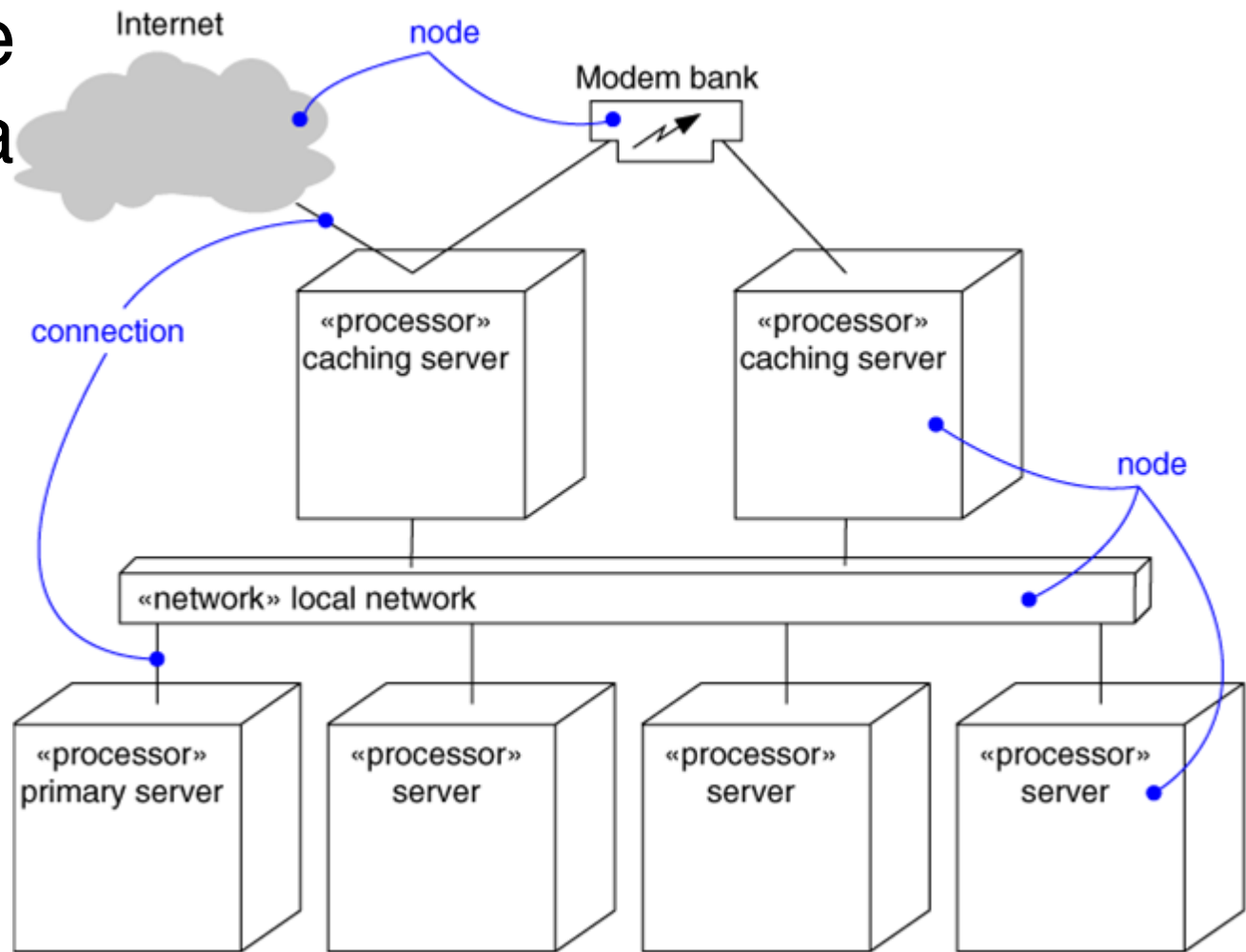
Component Diagram

- Captures the physical structure of the implementation
- Built as part of architectural specification
- Purpose
 - Organize source code
 - Construct an executable release
 - Specify a physical database
- Developed by architects and programmers



Deployment Diagram

- Captures the topology of a system's hardware



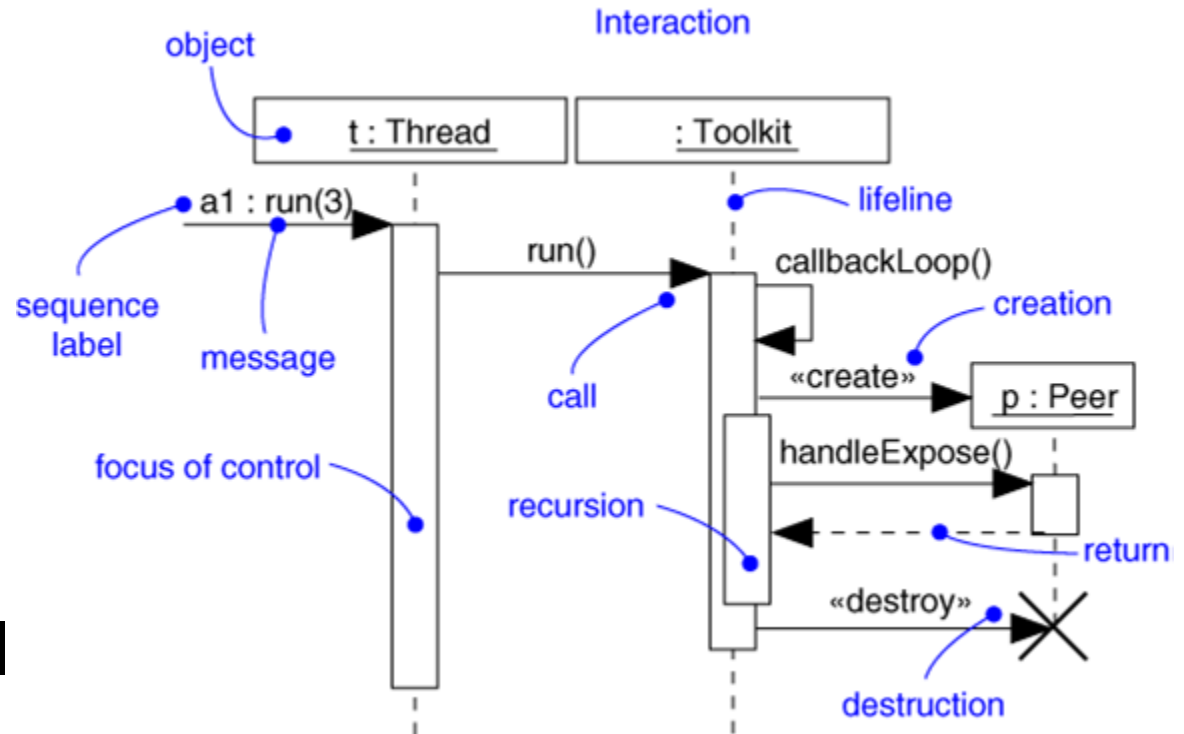
Deployment Diagram

- Captures the topology of a system's hardware
- Built as part of architectural specification
- Purpose
 - Specify the distribution of components
 - Identify performance bottlenecks
- Developed by architects, networking engineers, and system engineers



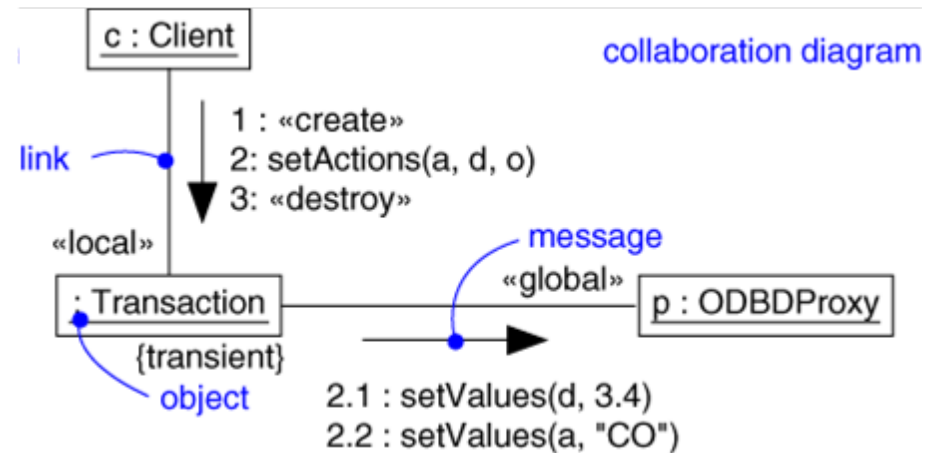
Sequence Diagram

- Captures dynamic behavior (time-oriented)
- Purpose
 - Model flow of control
 - Illustrate typical scenarios



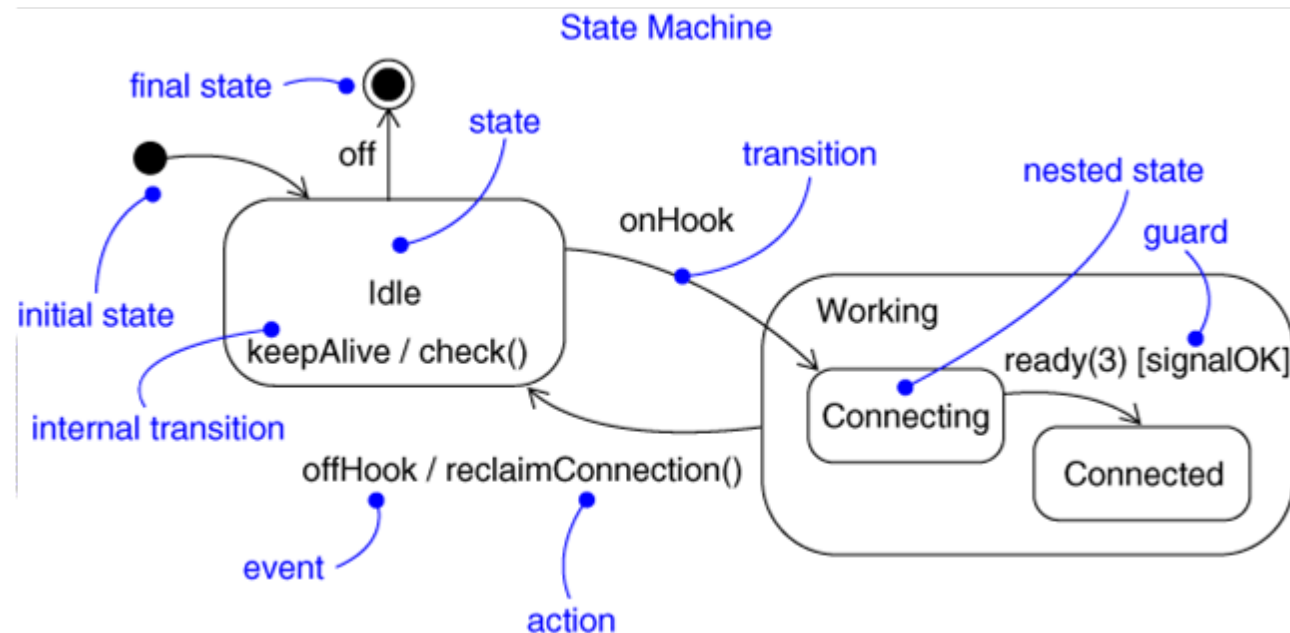
Collaboration Diagram

- Captures dynamic behavior (message-oriented)
- Purpose
 - Model flow of control
 - Illustrate coordination of object structure and control



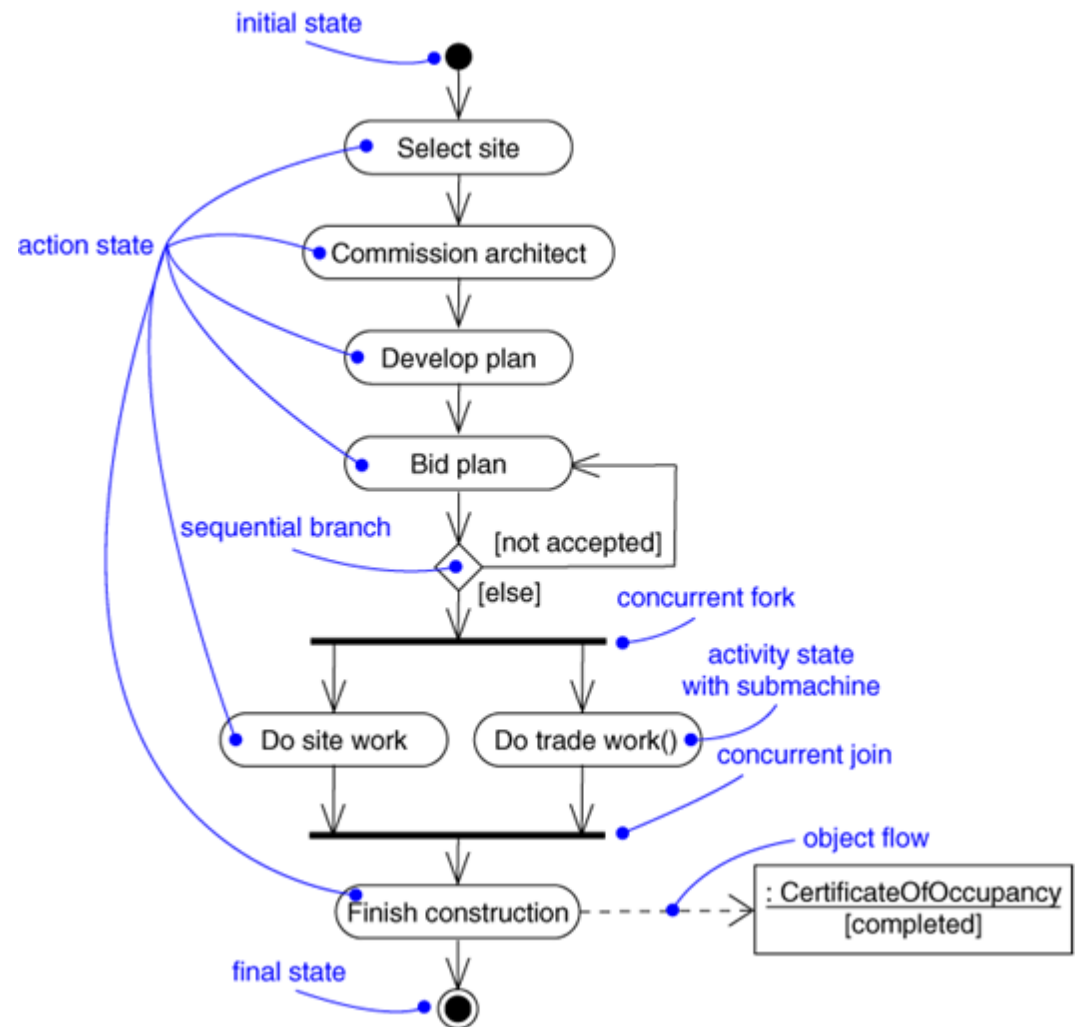
Statechart Diagram

- Captures dynamic behavior (event-oriented)
- Purpose
 - Model object lifecycle
 - Model reactive objects (user interfaces, devices, etc.)

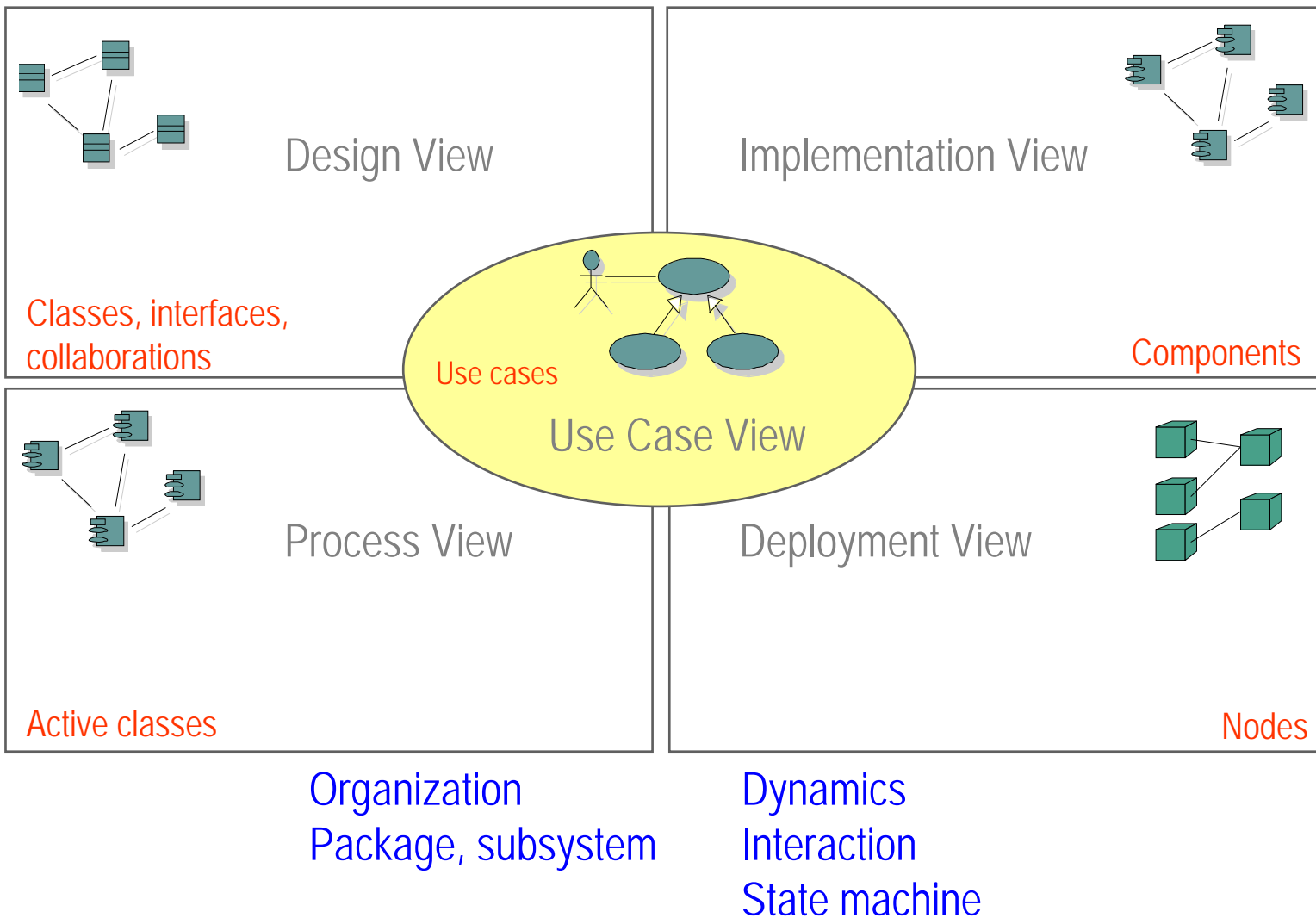


Activity Diagram

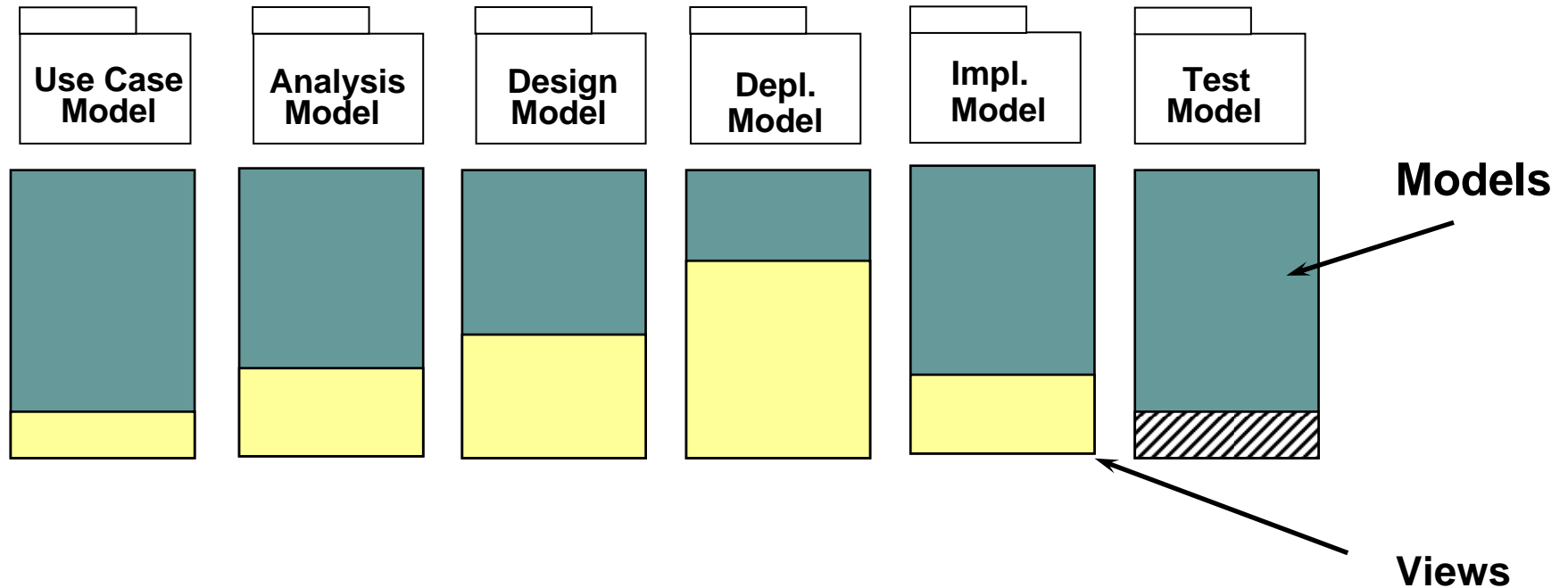
- Captures dynamic behavior (activity-oriented)
- Purpose
 - Model business workflows
 - Model operations



Architecture and the UML



Architecture and Models



Architecture embodies a collection of views of the models

Chương 5

NẮM BẮT YÊU CẦU HĐT

- Các artifacts cần tạo ra
- Các workers tham gia
- Qui trình nắm bắt yêu cầu

Mục đích của hoạt động nắm bắt yêu cầu

Mục đích của hoạt động nắm bắt yêu cầu là xây dựng mô hình hệ thống mà sẽ được xây dựng bằng cách dùng các use-case. Các điểm bắt đầu cho hoạt động này khá đa dạng :

- từ mô hình nghiệp vụ (business model) cho các ứng dụng nghiệp vụ.
- từ mô hình lĩnh vực (domain model) cho các ứng dụng nhúng (embeded).
- từ đặc tả yêu cầu của hệ thống nhưng được tạo bởi nhóm khác và/hoặc dùng các phương pháp đặc tả khác (thí dụ như hướng cấu trúc).
- từ 1 điểm nào đó nằm giữa các điểm xuất phát trên.

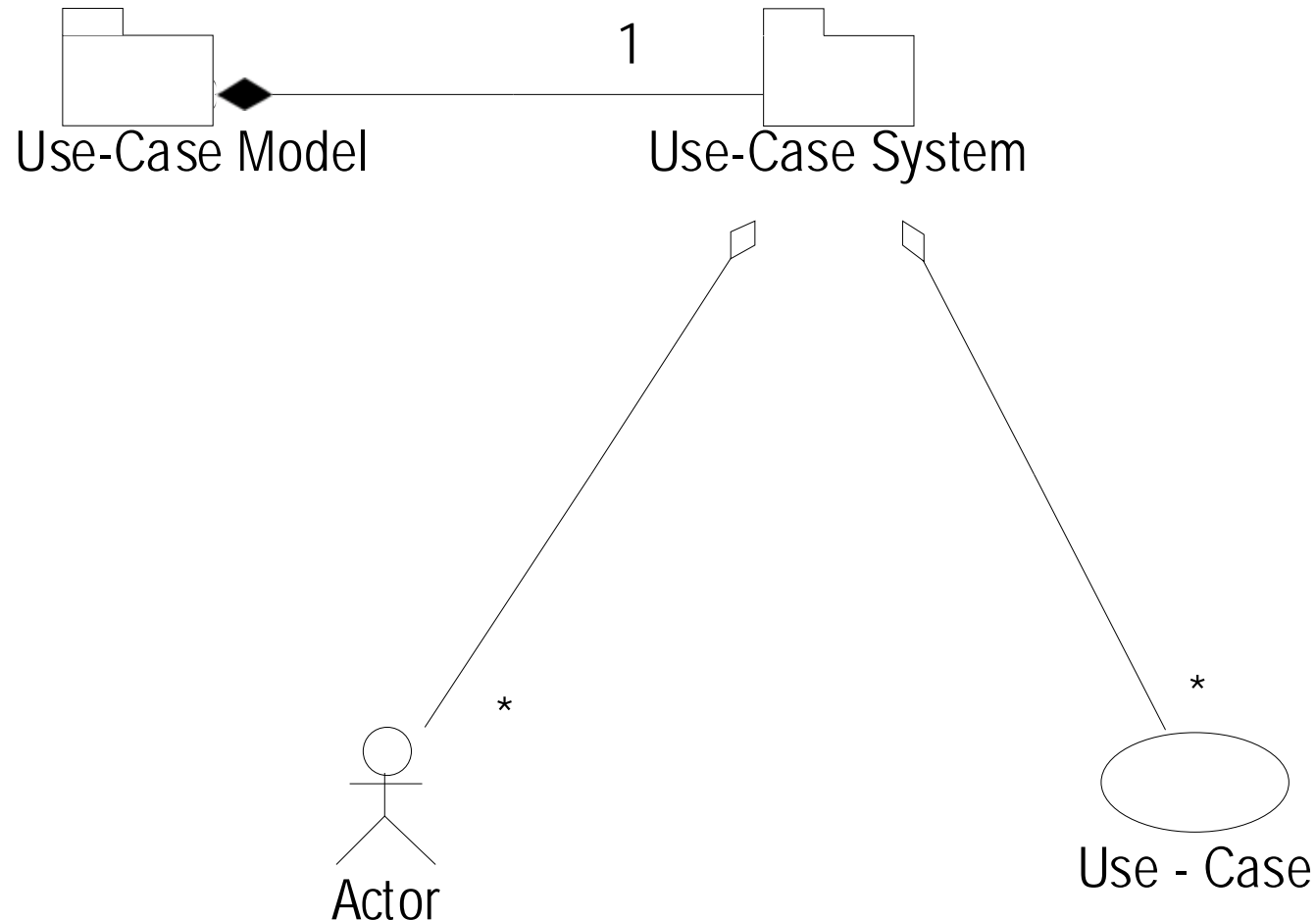
Các artifacts cần tạo ra trong nắm bắt yêu cầu



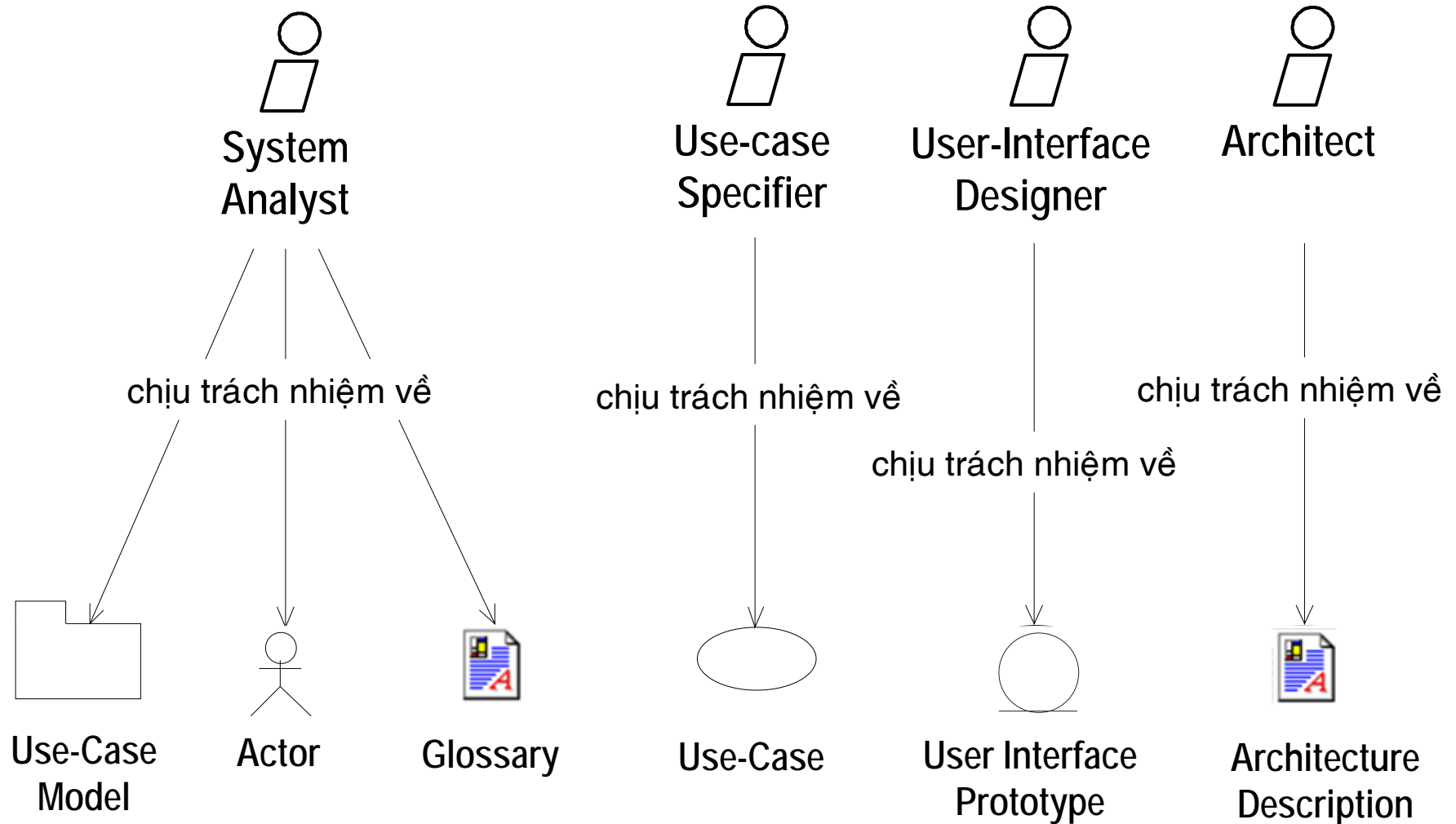
Mô hình use-case :

- actor : người/hệ thống ngoại/thiết bị ngoại tương tác với hệ thống
- use-case : các chức năng có nghĩa của hệ thống cung cấp cho actor.
 - flow of events
 - các yêu cầu đặc biệt của use-case
- đặc tả kiến trúc (view of use-case model)
- bảng thuật ngữ
- các prototype giao diện với user (user-interface prototype)

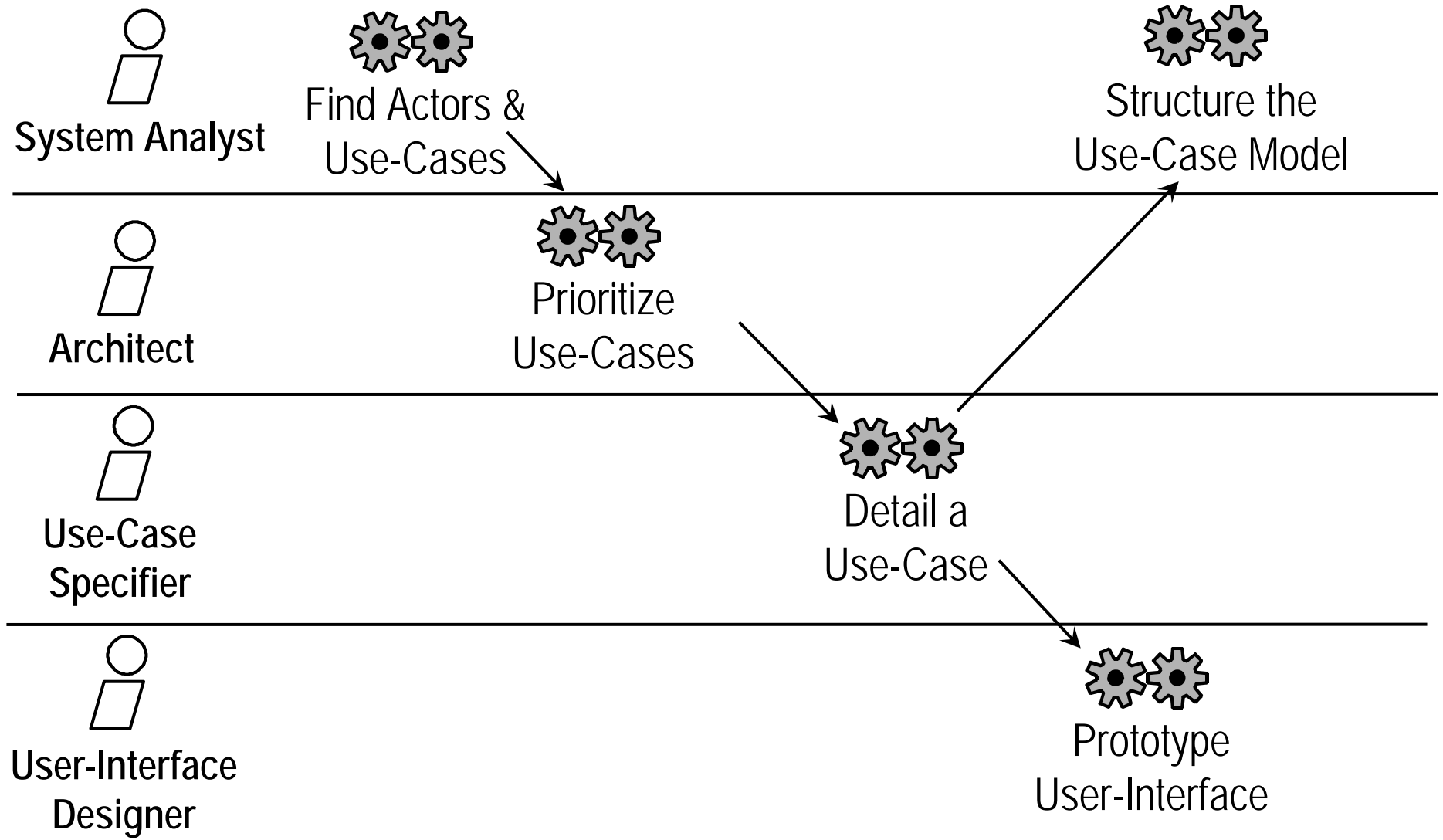
Các artifacts cần tạo ra trong nắm bắt yêu cầu



Các workers trong nắm bắt yêu cầu



Quy trình nắm bắt yêu cầu



Tìm Actors & Use cases

Mục đích nhận dạng actor và use-case là để :

- Giới hạn hệ thống với môi trường bao quanh nó.
- Phát họa ai và các gì sẽ tương tác với hệ thống và hệ thống cung cấp các chức năng gì.
- Nắm bắt và định nghĩa danh sách các thuật ngữ chung thiết yếu cho việc tạo các đặc tả về các chức năng của hệ thống.

Hoạt động này gồm 4 bước :

- Tìm các actor của hệ thống.
- Tìm các use cases của hệ thống.
- Miêu tả vắn tắt về từng use-case.
- Miêu tả toàn thể mô hình use-case.

Tìm Actors

Việc tìm các actor phụ thuộc vào điểm xuất phát : nếu xuất phát từ mô hình nghiệp vụ hay mô hình lĩnh vực thì việc tìm actor rất đơn giản. Còn nếu xuất phát từ các ý niệm mơ hồ thì hãy trả lời các câu hỏi sau :

- Ai là người sử dụng chức năng chính của hệ thống ?
- Ai cần sự hỗ trợ từ hệ thống để thực hiện công việc thường nhật của họ ?
- Ai phải thực hiện công việc bảo dưỡng, quản trị và giữ cho hệ thống hoạt động ?
- Hệ thống sẽ kiểm soát thiết bị phần cứng nào ?
- Hệ thống đang xây dựng cần tương tác với những hệ thống khác không ? Hệ thống nào ?
- Ai hoặc vật thể nào quan tâm đến hay chịu ảnh hưởng bởi kết quả mà hệ thống phần mềm tạo ra ?

Tìm Use-Cases

Việc tìm các use-case phụ thuộc vào điểm xuất phát : nếu xuất phát từ mô hình nghiệp vụ hay mô hình lĩnh vực thì việc tìm use-case rất đơn giản. Còn nếu xuất phát từ các ý niệm mơ hồ thì hãy trả lời các câu hỏi sau :

- *Actor* yêu cầu chức năng gì của hệ thống ?
- *Actor* cần phải đọc, tạo, xóa, sửa đổi hoặc lưu trữ thông tin nào của hệ thống ?
- *Actor* cần thiết phải được cảnh báo về những sự kiện trong hệ thống, hay *actor* cần phải báo hiệu cho hệ thống về vấn đề nào đó không ?
- Hệ thống có thể hỗ trợ một số công việc thường nhật của *actor* nào đó không ?

Miêu tả văn tắt từng Use-Cases

Mỗi use-case sau khi tìm được, nên được đặt tên, được miêu tả bằng vài câu tổng kết các hoạt động rồi sau đó đặc tả từng bước hệ thống cần gì để tương tác với actor.

Dùng lược đồ và các flow of events để miêu tả mô hình use-case tổng thể, đặc biệt là các mối quan hệ giữa các use-case và với actor.

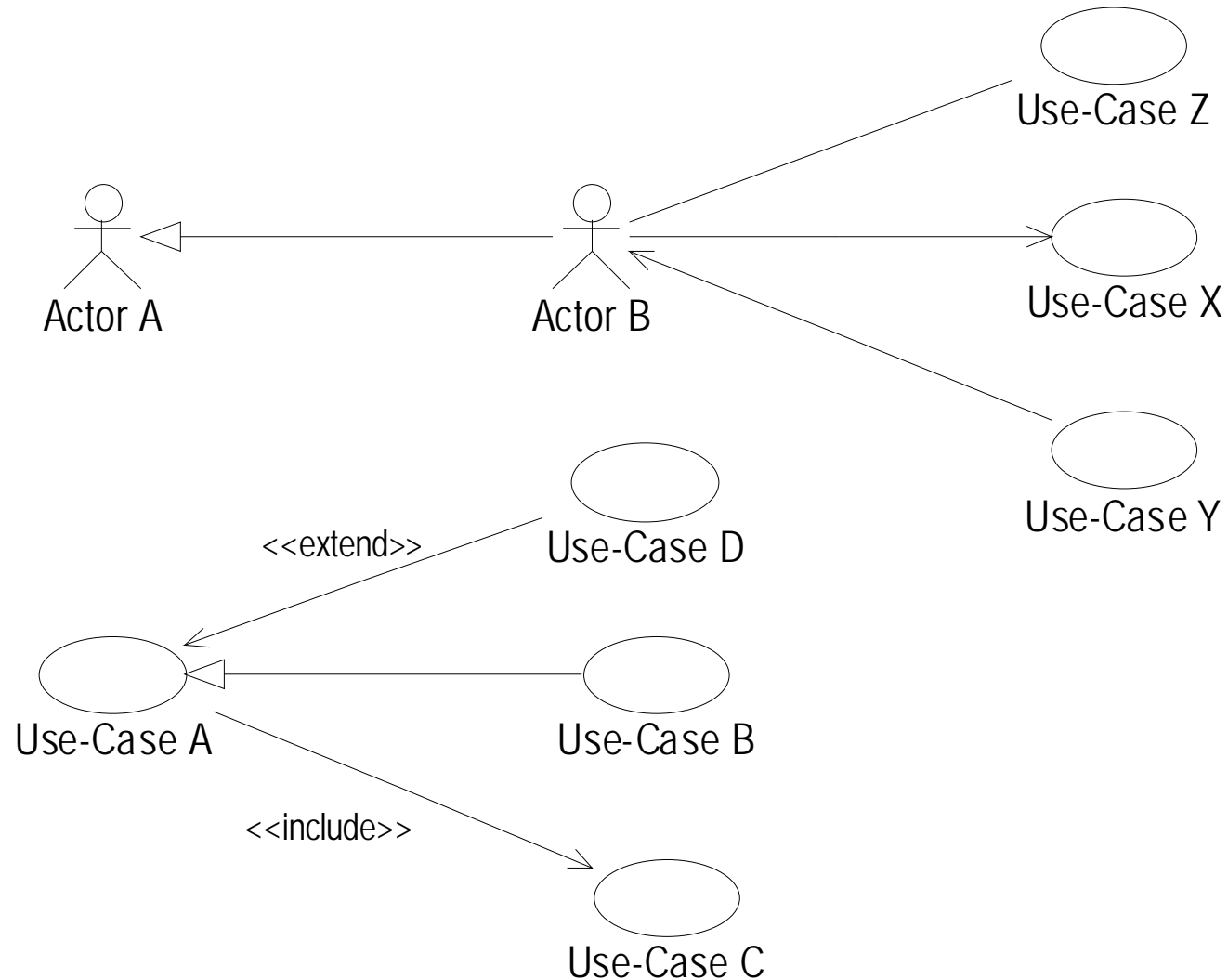
Quan hệ giữa các actors : tổng quát hóa (generalization).

Quan hệ giữa actor và use-case : liên kết (association).

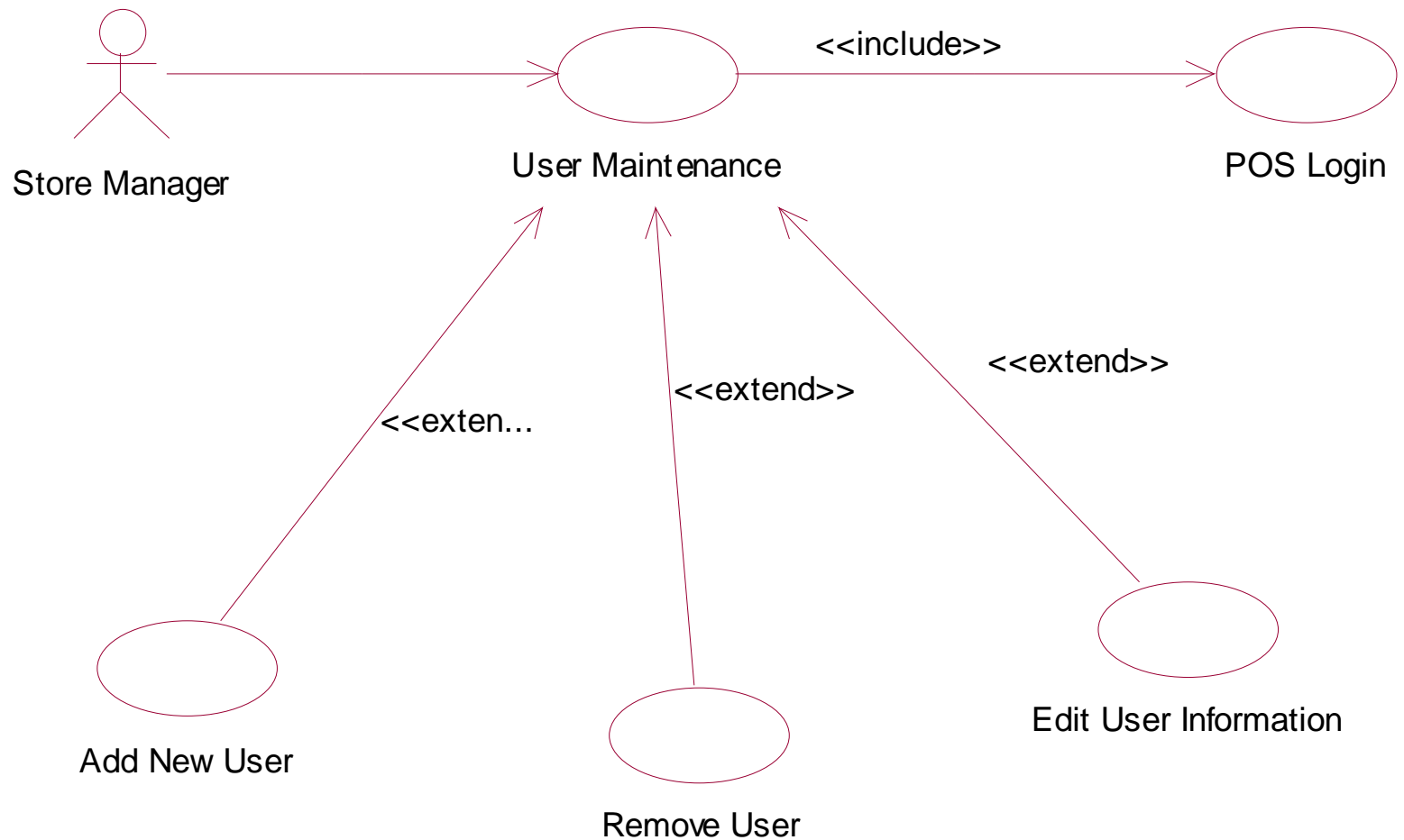
Quan hệ giữa các use-cases :

- tổng quát hóa.
- include
- extend

Các mối quan hệ giữa các actor và use-cases



Các mối quan hệ giữa các use-cases và use-cases

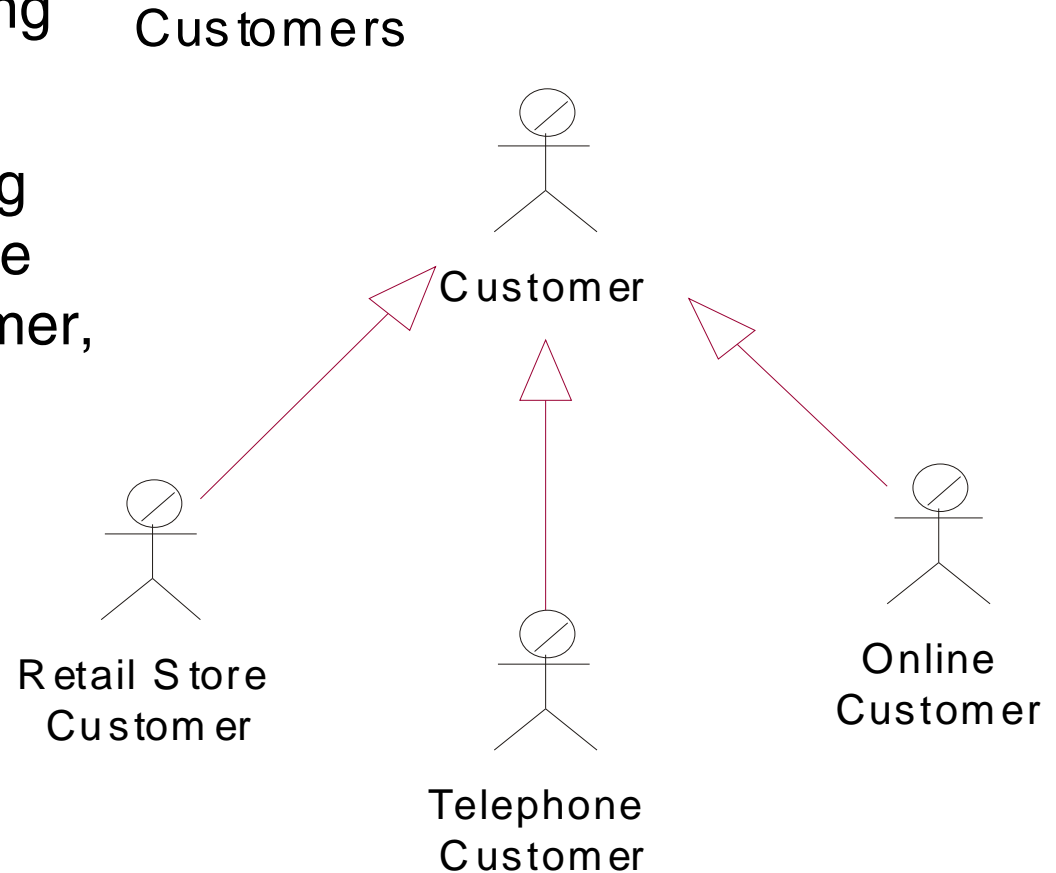


Các mối quan hệ giữa các actor và actor

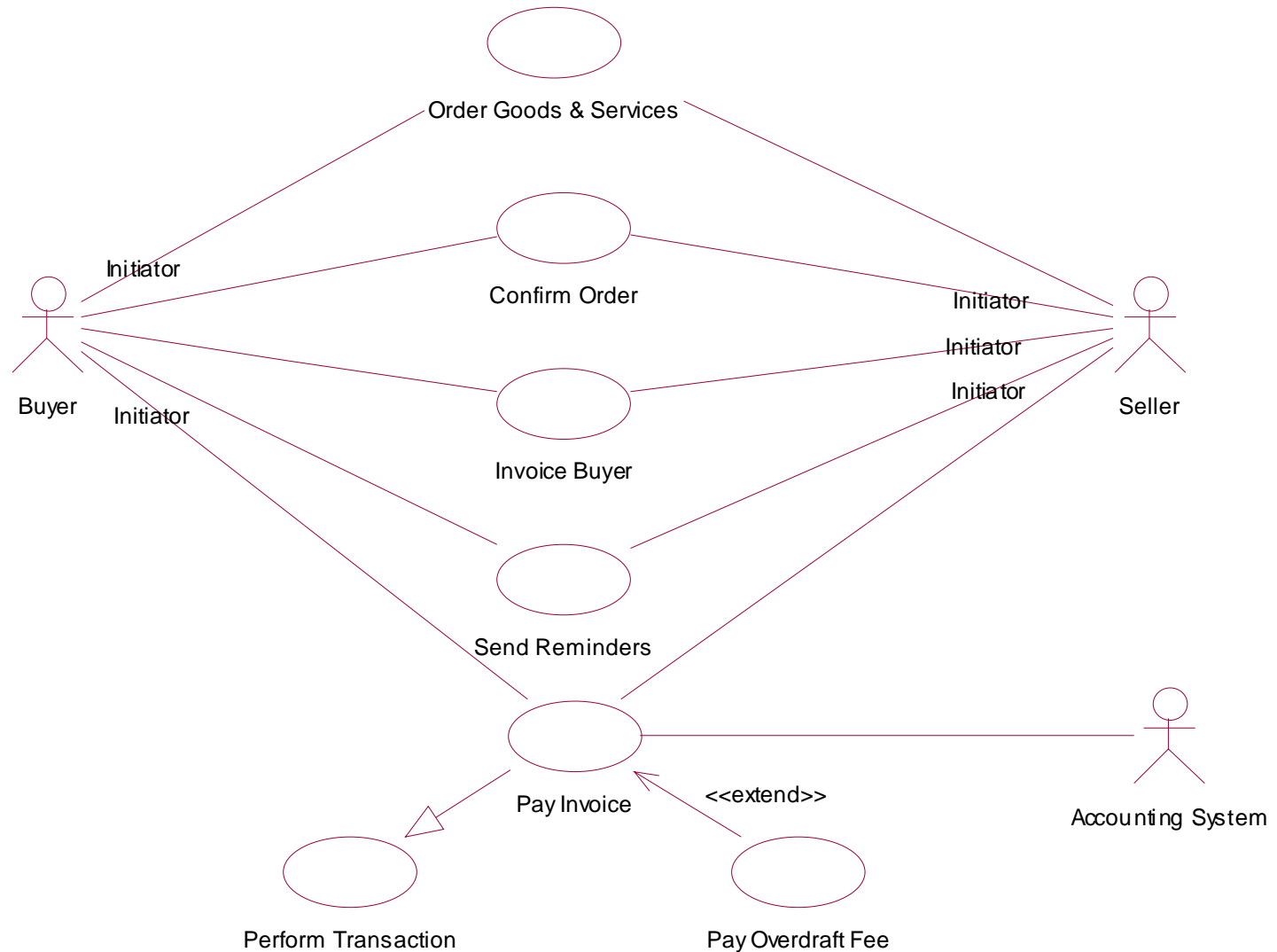
Quan hệ giữa các actors : tổng quát hóa (generalization).

Thí dụ Customer là actor tổng quát hóa của các actor Online Customer, Telephone Customer, Retail Store Customer.

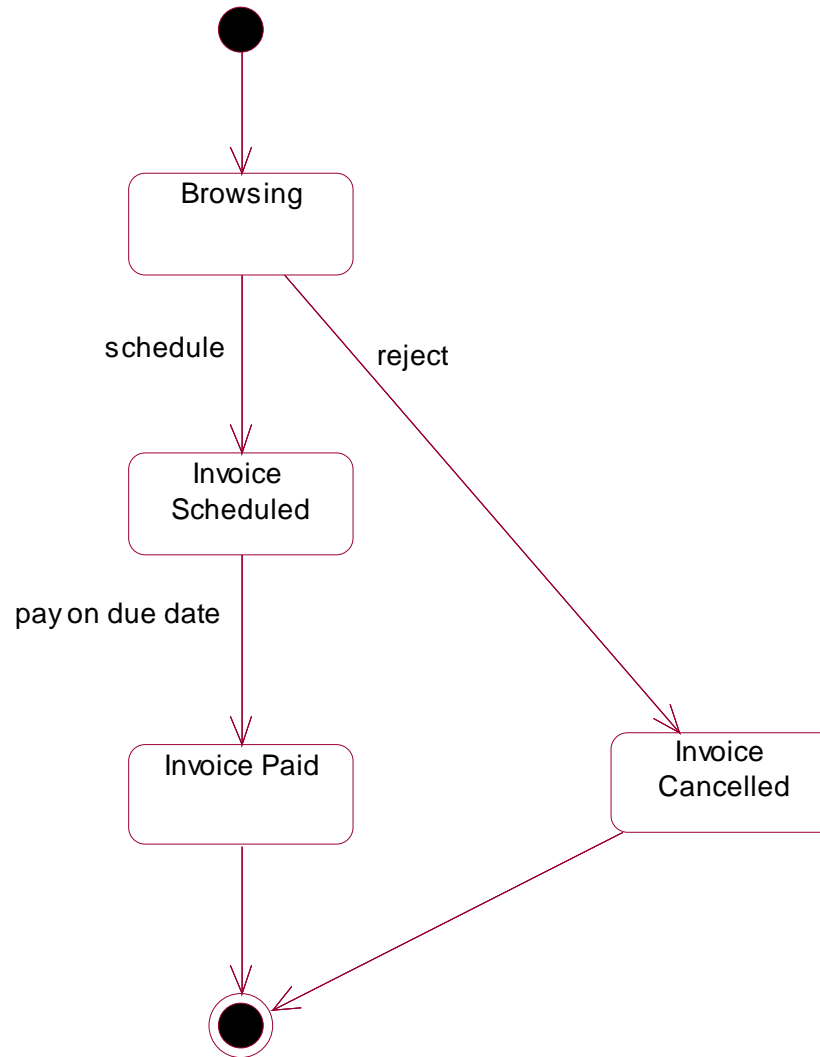
Lưu ý actor tổng quát hóa thường không có thật, nó là phần tử trừu tượng.



Lược đồ use-case Sales:From Order to Delivery



Lược đồ trạng thái của use-case Pay Invoice



Miêu tả tổng thể mô hình Use-Cases

Xây dựng các lược đồ use-case và các đặc tả giải thích mô hình use-case, nhất là cách thức mà các use-case quan hệ với nhau hay với các actor :

- lược đồ miêu tả các use-case phục vụ cho 1 actor hay 1 use-case nghiệp vụ.
- để đảm bảo tính nhất quán khi miêu tả nhiều use-case đồng thời, nên xây dựng 1 bảng thuật ngữ chung (glossary).
- mô hình use-case có thể được tổ chức dạng cây thứ bậc nhờ các package use-case.
- xây dựng đặc tả "survey" cho mô hình use-case tổng thể và nhờ khách hàng và người dùng kiểm tra, đánh giá lại.

Sắp thứ tự ưu tiên các use-case

- các use-case tìm được không phải thiết yếu như nhau, do đó kiến trúc sư cần sắp xếp thứ tự ưu tiên chúng để xác định use-case nào nên được phát triển trước, use-case nào được phát triển sau.
- kết quả của hoạt động này là xây dựng được góc nhìn kiến trúc của mô hình use-case, nó được dùng để hoạch định các bước lập cùng với các yếu tố khác như nghiệp vụ, kinh tế...

Chi tiết hóa Use-Case

Mục đích là đặc tả "flow of events" cho từng use-case :

- cấu trúc đặc tả use-case.
- đặc tả use-case bao gồm những gì.
- hình thức hóa đặc tả use-case.

Cấu trúc đặc tả use-case :

- gồm 1 luồng công việc cơ bản và các luồng phụ.

Các luồng phụ có thể xảy ra vì các lý do :

- Actor có thể chọn thực hiện 1 trong nhiều nhánh.
- Nếu hơn 1 actor dùng use-case, các hoạt động của họ có thể ảnh hưởng lẫn nhau.
- Hệ thống có thể phát hiện lỗi nhập từ actor.
- 1 số tài nguyên không hoạt động tốt làm cho use-case không hoàn tất công việc đúng của nó.

Đặc tả use-case gồm những gì ?

- nên định nghĩa trạng thái bắt đầu.
- khi nào và cách nào use-case bắt đầu.
- thứ tự các hoạt động được thực hiện.
- khi nào và cách nào use-case kết thúc.
- nên định nghĩa trạng thái kết thúc.
- không cho phép nhiều 'path' thực thi.
- Có thể miêu tả luồng thi hành phụ trong đặc tả luồng cơ bản.
- Đặc tả luồng phụ được rút trích từ luồng cơ bản.
- Tương tác giữa hệ thống và actor và chúng trao đổi những gì.
- Việc dùng các đối tượng, giá trị, tài nguyên trong hệ thống.
- Phải miêu tả rõ ràng hệ thống làm gì và actor làm gì.

Hình thức hóa use-case (Formalizing)

Khi sự tương tác giữa actor và use-case gồm nhiều trạng thái phức tạp ta nên dùng kỹ thuật mô hình trực quan để diễn tả use-case vì nó giúp nhà phân tích hiểu rõ hơn về use-case :

- lược đồ trạng thái UML có thể được dùng để miêu tả trạng thái của use-case và sự chuyển giữa các trạng thái.
- lược đồ hoạt động có thể được dùng để miêu tả sự chuyển trạng thái chi tiết hơn dưới dạng các hoạt động.
- lược đồ tương tác có thể được dùng để miêu tả các tương tác giữa đối tượng use-case và đối tượng actor.

Không nên lạm dụng các lược đồ vì đây là ngôn ngữ của nhà phát triển, các khách hàng và người dùng khó lòng hiểu nổi.

Cấu trúc lại mô hình Use-Case

Mô hình use-case được cấu trúc lại để :

- rút trích các use-case tổng quát và dùng chung bởi các use-case đặc biệt hơn.
- rút trích các use-case nhiệm ý và phụ thêm để nói rộng use-case khác.

Trước khi hoạt động này xảy ra :

- nhà phân tích đã nhận diện tương đối đầy đủ các actor và use-case, miêu tả chúng trong các lược đồ để cấu thành mô hình use-case tổng thể.
- người đặc tả use-case đã phát triển đặc tả chi tiết cho mỗi use-case.

Cấu trúc lại mô hình Use-Case

Các công việc cụ thể :

- Nhận dạng các use-case tổng quát được dùng chung.
- Nhận dạng các use-case có quan hệ "extend".
- Nhận dạng các use-case có quan hệ "include".

Một số điều lưu ý :

- Cấu trúc các use-case và mối quan hệ giữa chúng nên phản ánh các chức năng thực tế.
- Mỗi use-case cần được xử lý như 1 artifact riêng biệt, do đó không nên chọn use-case quá lớn hay quá nhỏ.
- Tránh chia nhỏ use-case.

Chương 6

PHÂN TÍCH HƯỚNG ĐỐI TƯỢNG

- Các artifacts cần tạo ra
- Các workers tham gia
- Qui trình phân tích

Mục đích của phân tích yêu cầu

Mục đích của hoạt động phân tích yêu cầu là xây dựng mô hình phân tích với các đặc điểm sau :

- dùng ngôn ngữ của nhà phát triển để miêu tả mô hình.
- thể hiện góc nhìn từ bên trong của hệ thống.
- được cấu trúc từ các class phân tích và các package phân tích.
- được dùng chủ yếu bởi nhà phát triển để hiểu cách thức tạo hình dạng hệ thống.
- loại trừ mọi chi tiết dư thừa, không nhất quán.
- phát họa các hiện thực cho các chức năng bên trong hệ thống.
- định nghĩa các dẫn xuất use-case, mỗi dẫn xuất use-case cấp phân tích miêu tả sự phân tích 1 use-case.

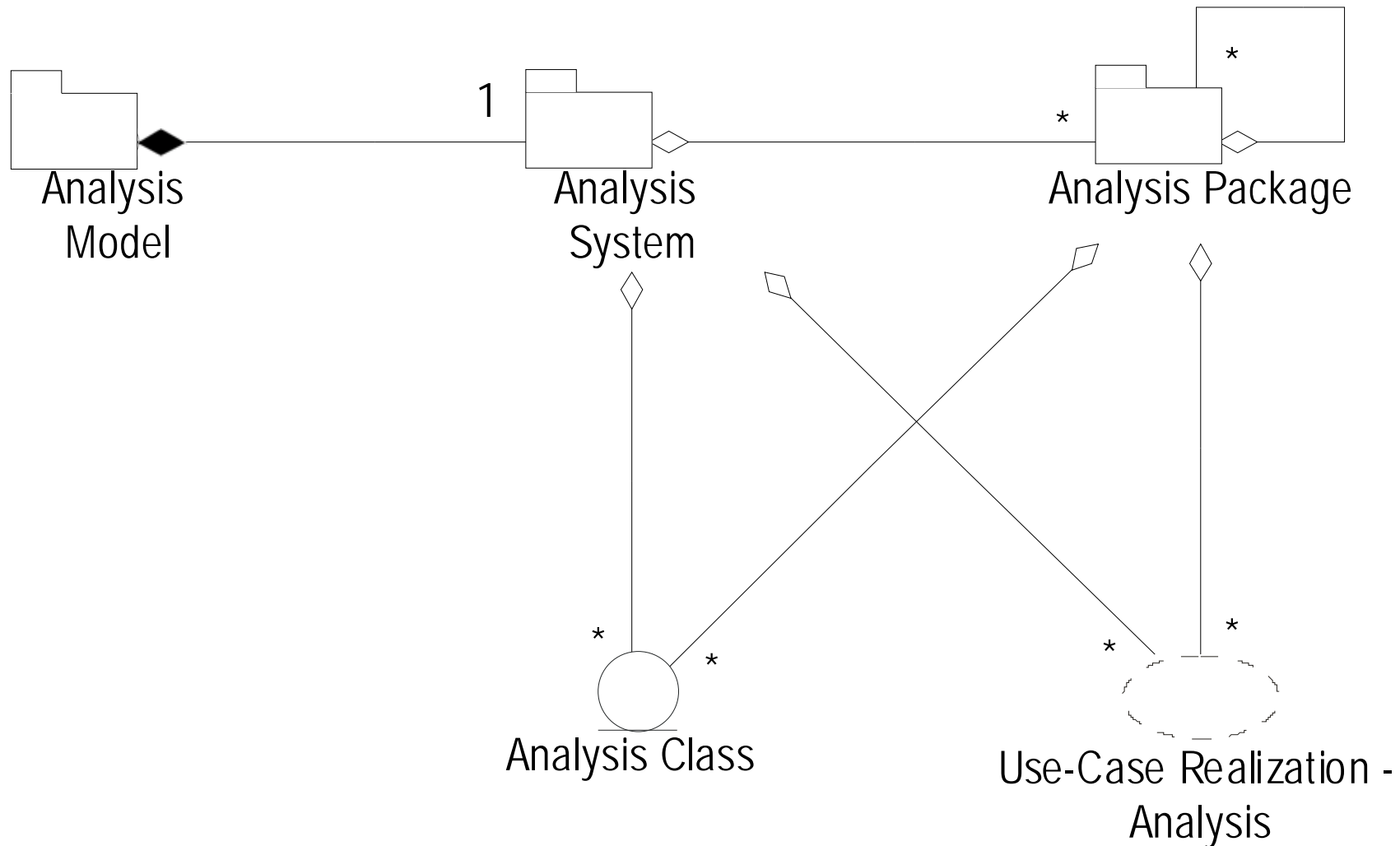
Các artifacts cần tạo ra trong phân tích yêu cầu



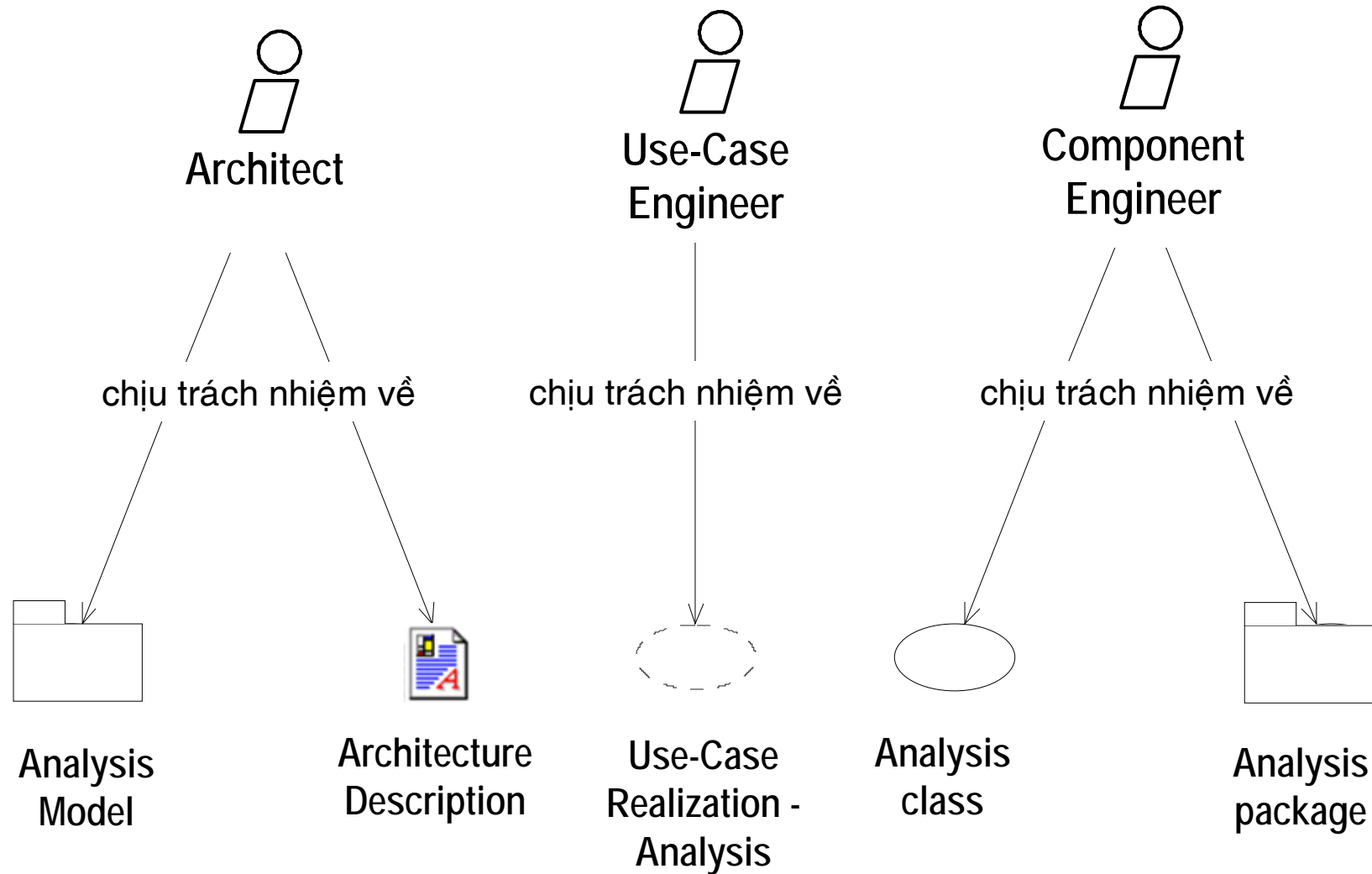
Mô hình phân tích = hệ thống phân tích :

- các class phân tích
 - boundary class
 - entity class.
 - control class
- các dẫn xuất use-case cấp phân tích :
 - các lược đồ class phân tích
 - các lược đồ tương tác (cộng tác,...).
 - 'flow of events' ở cấp phân tích
 - các yêu cầu đặc biệt của use-case
- các package phân tích
- đặc tả kiến trúc (view of analysis model)

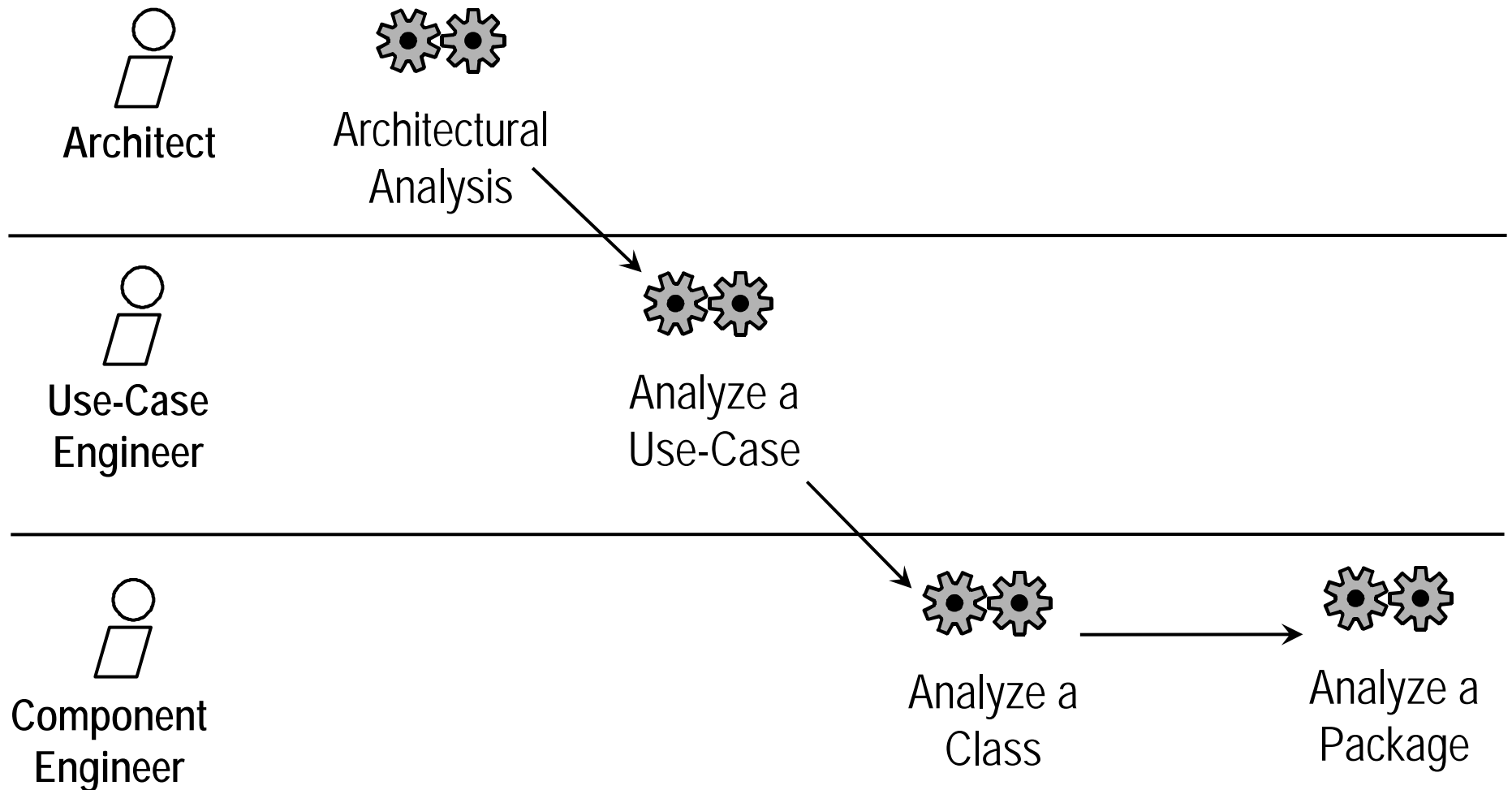
Các artifacts cần tạo ra trong phân tích yêu cầu



Các workers trong phân tích yêu cầu



Quy trình phân tích yêu cầu



Phân tích kiến trúc : nhận dạng các package phân tích

Mục đích của phân tích kiến trúc là phát họa mô hình phân tích và kiến trúc hệ thống bằng cách nhận dạng các package phân tích, các class phân tích dễ thấy và các yêu cầu đặc biệt chung cho hệ thống.

Các package phân tích giúp tổ chức hệ thống thành những đơn vị nhỏ dễ quản lý. Mỗi package chứa 1 số use-case với tính chất sau :

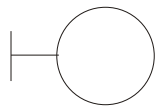
- các use-case hỗ trợ cho cùng 1 qui trình nghiệp vụ.
- các use-case hỗ trợ cho cùng 1 actor.
- các use-case có quan hệ lẫn nhau : tổng quát hóa, include và extend.

Theo thời gian, khi việc phân tích tiến triển, sự tinh chế cấu trúc các package sẽ tiến triển theo.

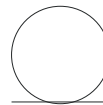
3 loại class phân tích

Có 3 loại (stereotype) class phân tích :

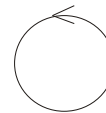
- class biên (boundary class) mô hình sự tương tác giữa actor và hệ thống
- class thực thể (entity class) mô hình thông tin cần cho hệ thống, loại thông tin có tính bền vững, tồn tại lâu dài.
- class điều khiển (control class) mô hình việc xử lý, cộng tác, giao tác trong use-case.



Boundary class



Entity class



Control class

Phân tích kiến trúc : nhận dạng các class thực thể dễ thấy

Từ các class lĩnh vực hay các class nghiệp vụ trong bước nắm bắt yêu cầu, đề nghị 1 số class thực thể quan trọng nhất (từ 10-20).

Các class phân tích còn lại sẽ được nhận dạng trong hoạt động phân tích use-case.

Các yêu cầu đặc biệt cũng được nhận dạng để được xử lý trong các bước sau, chúng gồm :

- tính bền vững.
- sự phân tán & đồng thời.
- các tính chất an toàn dữ liệu.
- đề kháng với lỗi.
- quản lý giao tác.

Tính chất của mỗi yêu cầu đặc biệt sẽ được cân nhắc sau trong từng class và từng dẫn xuất use-case.

Phân tích use-case

Phân tích use-case là để :

- nhận dạng các class phân tích có đối tượng của chúng tham gia vào việc thực hiện 'flow of events' của use-case.
- phân phối hành vi của use-case bằng cách cho các đối tượng phân tích tương tác nhau.
- nắm bắt 1 số yêu cầu đặc biệt cho dẫn xuất use-case.



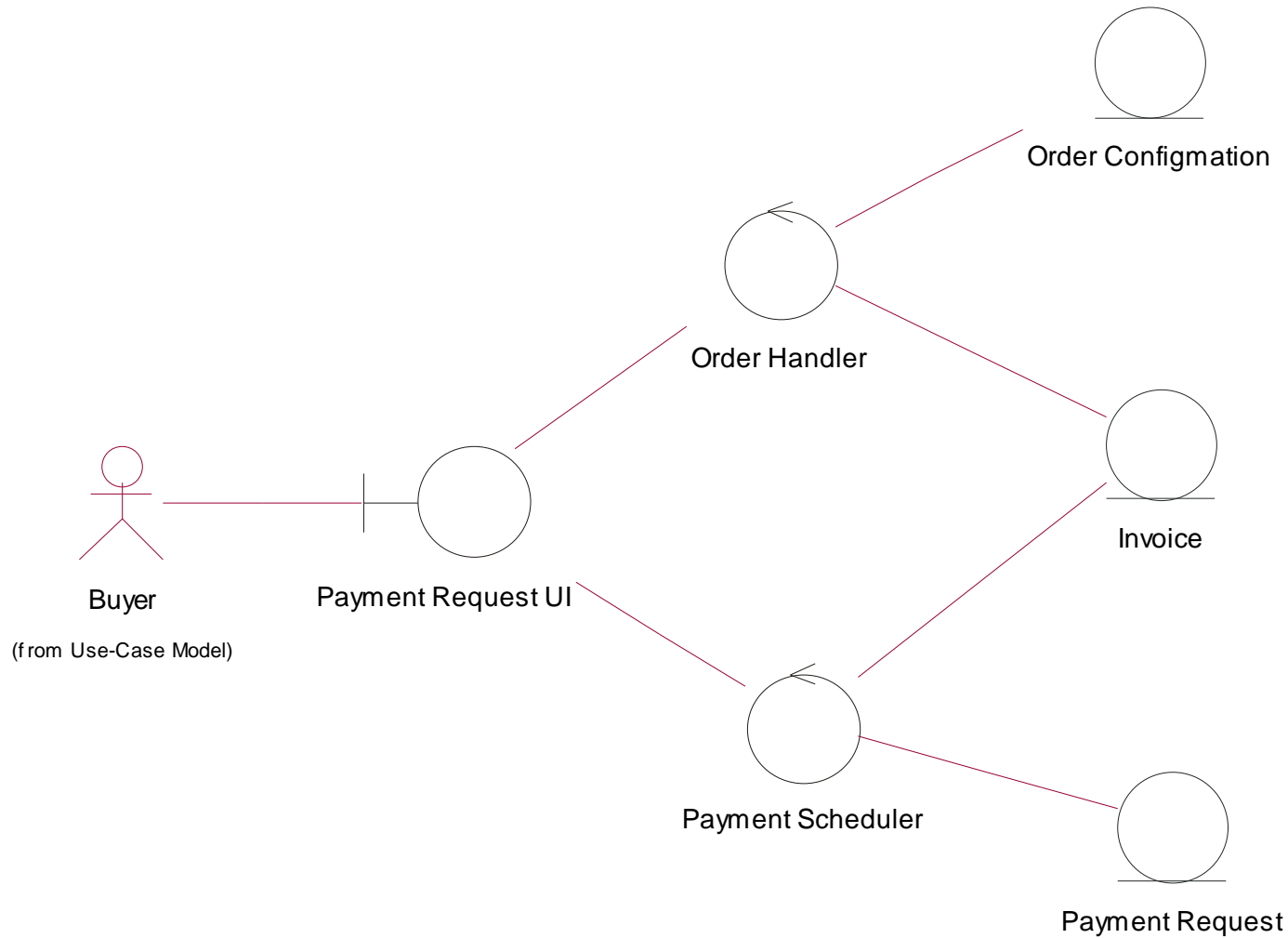
Phân tích use-case : nhận dạng các class phân tích

Trong bước này ta nhận dạng các class điều khiển, biên, thực thể cần thiết để hiện thực use-case và phát họa tên, trách nhiệm, thuộc tính và các mối quan hệ giữa chúng. Dùng các hướng dẫn sau :

- nhận dạng các class thực thể bằng cách chú ý các thông tin trong đặc tả use-case và trong mô hình lĩnh vực.
- nhận dạng class biên cơ sở cho mỗi class thực thể vừa tìm được.
- nhận dạng class biên trung tâm cho mỗi actor là con người.
- nhận dạng class biên trung tâm cho mỗi actor là hệ thống ngoại hay thiết bị I/O.
- nhận dạng class điều khiển có trách nhiệm xử lý trong dẫn xuất use-case.

Tập hợp các class phân tích tham gia vào dẫn xuất use-case thành 1 (hay nhiều) lược đồ class.

Thí dụ về lược đồ class phân tích cho use-case Pay Invoice



Phân tích use-case : miêu tả sự tương tác giữa các đối tượng phân tích

Cần chú ý các điểm sau trong lược đồ cộng tác :

- p. tử actor gửi 1 thông báo đến class biên để kích hoạt use-case.
- mỗi class phân tích nên có ít nhất 1 đối tượng tham gia vào lược đồ cộng tác.
- chưa vội kết hợp tác vụ cụ thể cho thông báo.
- các mối nối trong lược đồ cộng tác thường là 'instance' của mối quan hệ kết hợp giữa các class tương ứng.
- chưa tập trung vào thứ tự thời gian các thông báo.
- Lược đồ cộng tác nên xử lý tất cả mối quan hệ của use-case được hiện thực.
- cần bổ sung đặc tả dạng văn bản vào lược đồ cộng tác, đặc tả này nên được để vào 'flow of events cấp phân tích'.

Lược đồ cộng tác

Cần chú ý các điểm sau trong lược đồ cộng tác :

- các thông điệp được đánh số theo kiểu phân cấp.
 - 3.4.2 xảy ra sau 3.4.1 và cả 2 được lồng trong 3.4
 - 3.4.3a và 3.4.3b xảy ra đồng thời và được lồng trong 3.4
- cú pháp tổng quát của 1 thông điệp :

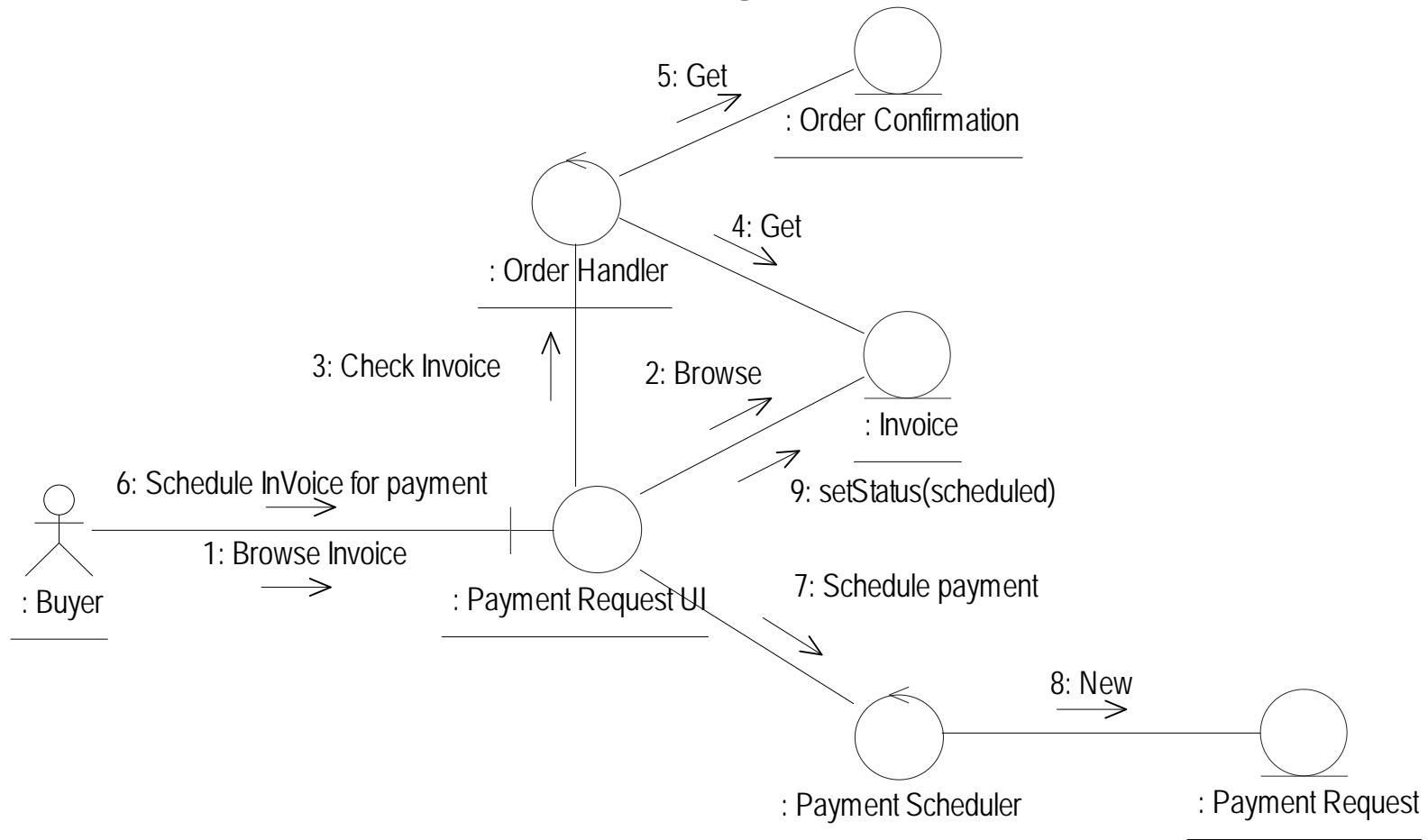
**predecessor guard-condition sequence-expression return-value :=
message-name argument-list**

Ví dụ :

- 2/ 1.3.1: p := find(specs)
- 1.1, 4.2/ 3.2 *[i:=1..6]: invert(x, color)

Lược đồ cộng tác

Các thành phần của lược đồ cộng tác :



Phân tích class

Mục đích của việc phân tích class là :

- nhận dạng và duy trì các nghĩa vụ, trách nhiệm của class phân tích dựa vào vai trò của nó trong dẫn xuất use-case.
- nhận dạng và duy trì các thuộc tính và các mối quan hệ của class phân tích.
- nắm bắt các yêu cầu đặc biệt liên quan đến việc hiện thực class phân tích.



Phân tích class : nhận dạng các nghĩa vụ

- tổ hợp các vai trò mà class đóng trong các dẫn xuất use-case khác nhau sẽ cho ta 1 số nghĩa vụ của class.
- nghiên cứu các lược đồ class và lược đồ tương tác trong các dẫn xuất use-case có class tham gia.
- đôi khi cần nghiên cứu 'flow of events cấp phân tích' của dẫn xuất use-case để tìm thêm các nghĩa vụ các class.

Phân tích class : nhận dạng các thuộc tính

Mỗi nghĩa vụ thường cần 1 số thuộc tính. Dùng các hướng dẫn sau :

- tên thuộc tính nên là danh từ.
- kiểu thuộc tính ở cấp phân tích nên ở cấp ý niệm, chưa cần kiểu cụ thể, nên dùng lại kiểu đã có khi đặc tả kiểu cho thuộc tính mới.
- nếu class phân tích quá phức tạp, nên tách 1 số thuộc tính phức tạp ra thành class riêng.
- thuộc tính của class thực thể thường dễ thấy.
- thuộc tính của class biên giao tiếp với người thường miêu tả thông tin được xử lý bởi user như các field text,...
- thuộc tính của class biên giao tiếp với hệ thống ngoài thường miêu tả các tính chất của giao tiếp.
- thuộc tính của class điều khiển ít khi có.
- đôi khi không cần các thuộc tính hình thức.

Phân tích class : nhận dạng mối quan hệ giữa các class

Các đối tượng tương tác nhau thông qua các lược đồ cộng tác. Các mối liên kết này thường là 'instance' của mối quan hệ kết hợp giữa các class. Các mối liên kết này cũng có thể ám chỉ nhu cầu về sự gộp nhiều đối tượng. Mối quan hệ gộp nên được dùng khi các đối tượng miêu tả :

- các khái niệm chứa vật lý khái niệm khác (xe chứa tài xế và khách)
- các khái niệm được xây dựng từ các khái niệm khác (xe gồm các bánh xe và động cơ).
- các khái niệm tạo thành tập hợp ý niệm nhiều đối tượng (gia đình gồm cha, mẹ và con).

Để rút trích các hành vi chung của nhiều class phân tích, ta có thể dùng class tổng quát hóa, nhưng chỉ nên ở cấp ý niệm.

Phân tích package

Mục đích của phân tích package là :

- đảm bảo từng package phân tích độc lập với các package khác nhiều như có thể có.
- đảm bảo package phân tích hoàn thành mục đích của nó là hiện thực 1 số class lĩnh vực hoặc 1 số use-case.
- miêu tả các phụ thuộc sao cho có thể ước lượng ảnh hưởng của các thay đổi trong tương lai.

Dùng các hướng dẫn sau :

- đảm bảo package chứa các class đúng, cố gắng cho tính kết dính cao bằng cách gộp các class có mối quan hệ chức năng.
- hạn chế tối đa sự phụ thuộc giữa các package, phân phối lại các class quá phụ thuộc vào package khác.

Chương 7

THIẾT KẾ HƯỚNG ĐỐI TƯỢNG

- Các artifacts cần tạo ra
- Các workers tham gia
- Quy trình thiết kế

Mục đích của thiết kế

Mục đích của công việc thiết kế là :

- đạt tới sự hiểu biết sâu sắc các vấn đề về các ràng buộc và các yêu cầu không chức năng có liên quan đến ngôn ngữ lập trình, sự dùng lại linh kiện, HĐH, công nghệ phân tán, đồng thời, database, giao diện, quản lý giao tác.
- tạo ra đầu vào cho hoạt động hiện thực bằng cách nắm bắt các hệ thống con, các interface và các class.
- chia công việc hiện thực ra nhiều phần để quản lý và xử lý bởi các đội khác nhau (có thể đồng thời).
- nắm bắt các interface chính giữa các hệ thống con.
- có thể hiển thị trực quan và xem xét bảng thiết kế dùng các ký hiệu chung.
- tạo ra mức trừu tượng của sự hiện thực hệ thống.

Các artifacts cần tạo ra trong thiết kế



Mô hình thiết kế = hệ thống thiết kế :

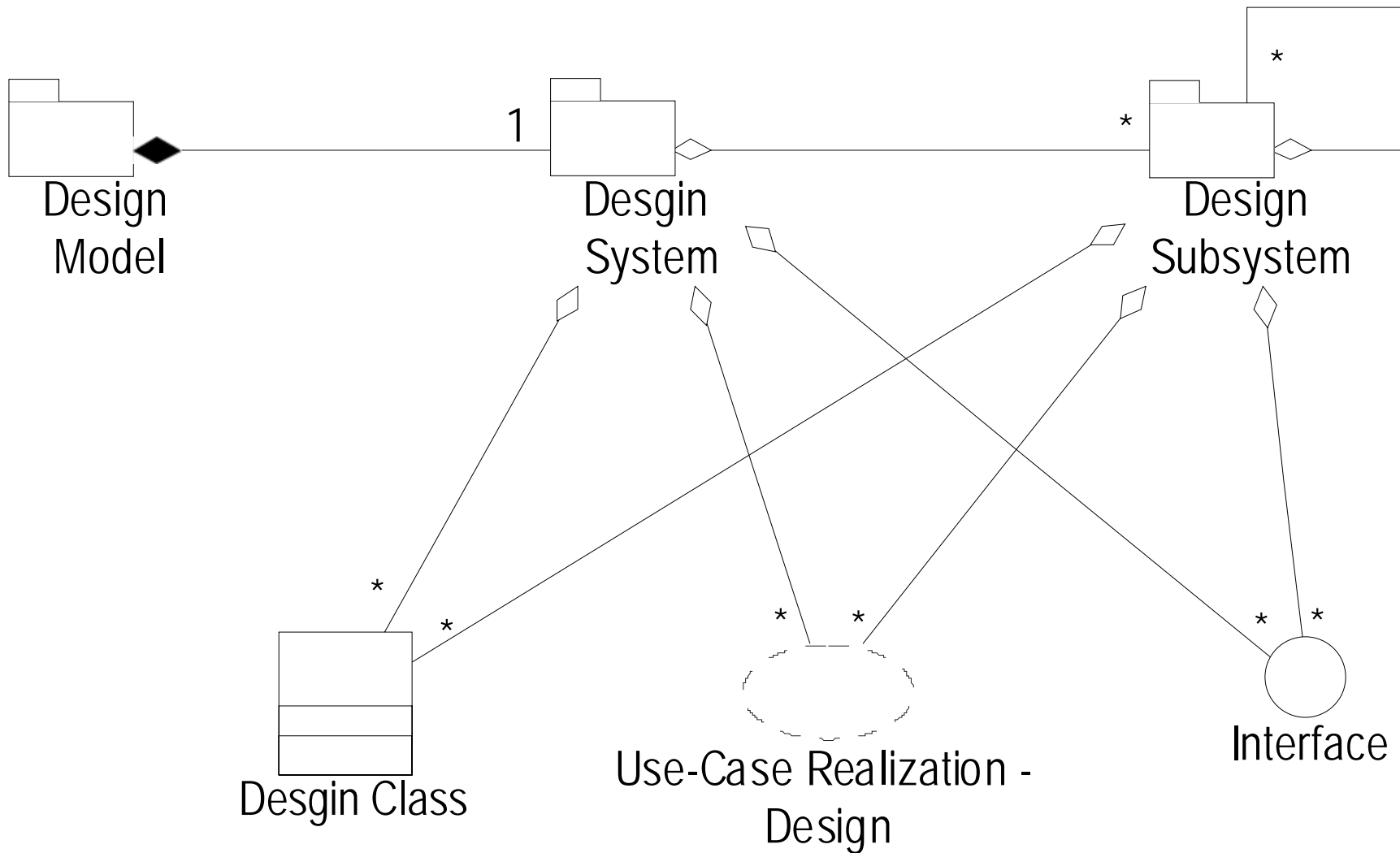
- các hệ thống con
- các class thiết kế.
- các interface của hệ thống con và class.
- các dẫn xuất use-case cấp thiết kế :
 - các lược đồ class
 - các lược đồ tương tác (trình tự, trạng thái,...).
 - 'flow of events' ở cấp thiết kế.
 - các yêu cầu cấp hiện thực.
- đặc tả kiến trúc (view of design model)



Mô hình bố trí :

- đặc tả kiến trúc (view of deployment model)

Các artifacts cần tạo ra trong thiết kế

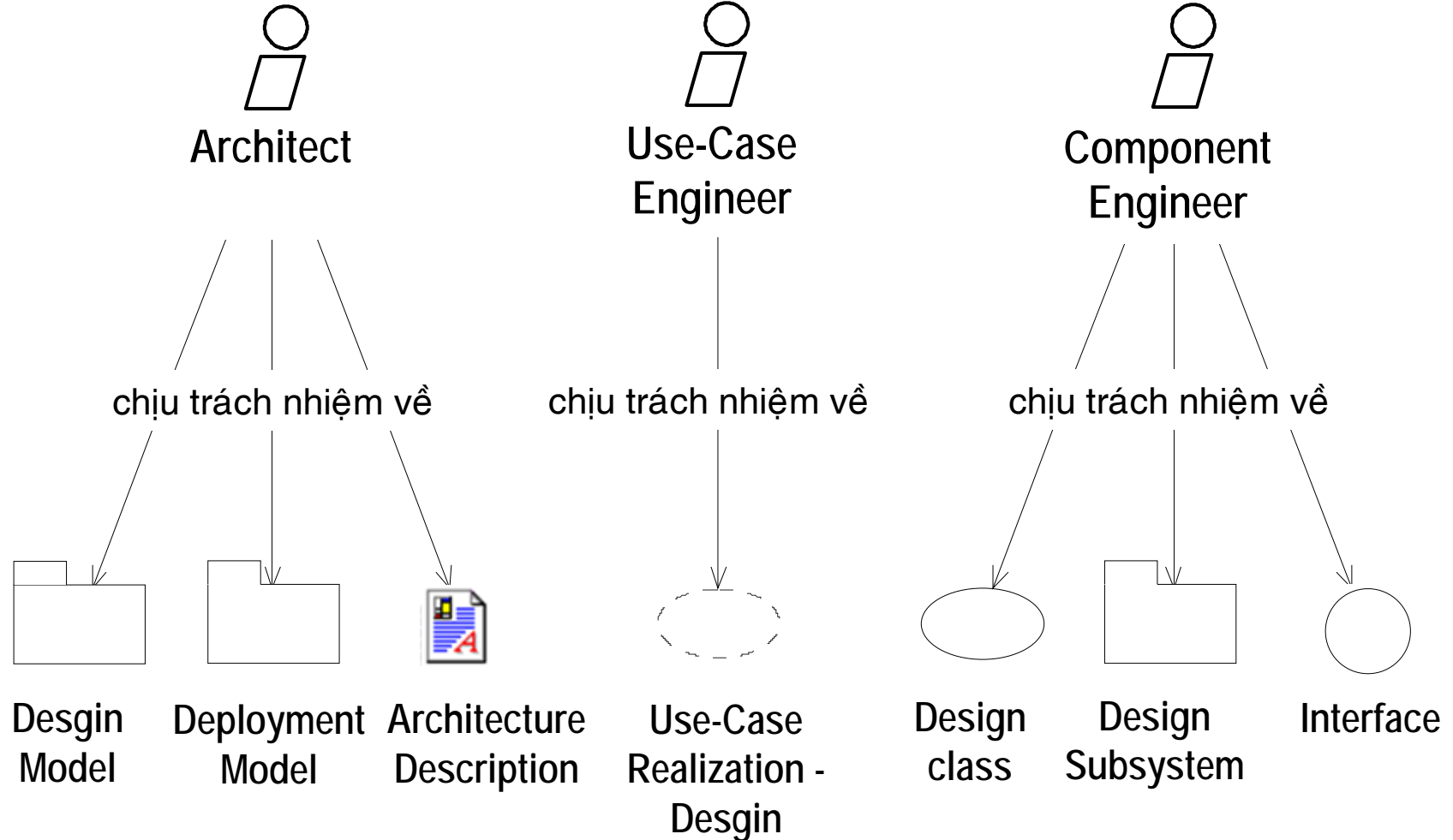


Đặc điểm của class thiết kế

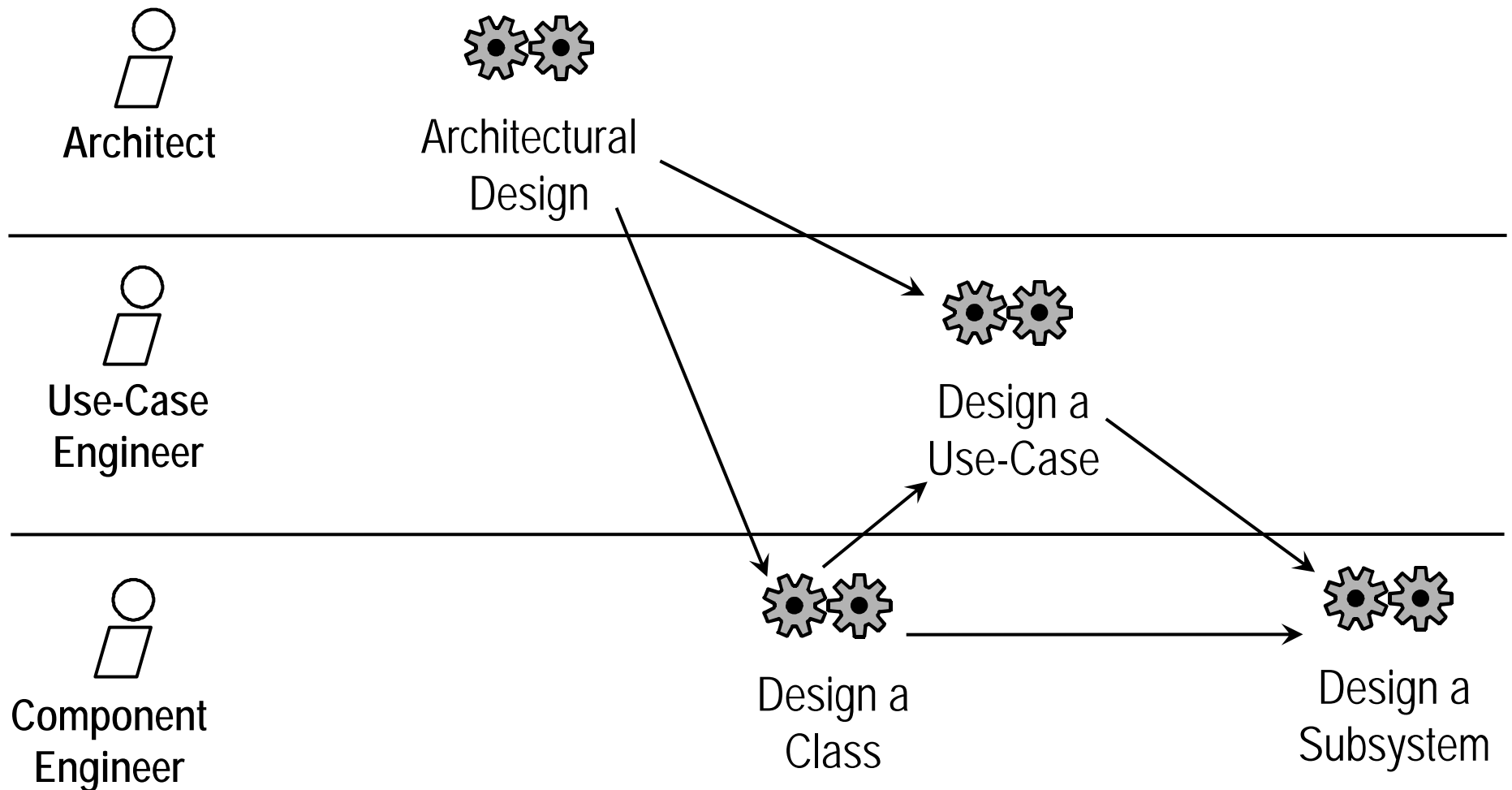
Class thiết kế là sự trừu tượng trực tiếp của class hiện thực :

- dùng ngôn ngữ lập trình để miêu tả class thiết kế.
- thường xác định tầm vực của các thành phần.
- các mối quan hệ có ý nghĩa trực tiếp tới hiện thực : tổng quát hóa → thừa kế, quan hệ gộp, kết hợp thành thuộc tính tương ứng.
- method trong thiết kế → method trong hiện thực.
- có thể delay việc xử lý 1 số yêu cầu tới lúc hiện thực.
- class thiết kế thường có stereotype tương ứng với ngôn ngữ lập trình : <<form>>, <<class module>>, <<user control>>...
- class thiết kế hiện thực (hay cung cấp) 1 interface.
- có thể có 1 số class active, nhưng nên tồn tại trong mô hình process thay vì trong mô hình thiết kế.

Các wokers trong hoạt động thiết kế



Quy trình thiết kế



Thiết kế kiến trúc : mục đích

Mục đích của thiết kế kiến trúc là phát họa mô hình thiết kế và mô hình bố trí cùng kiến trúc của chúng bằng cách nhận dạng các vấn đề sau :

- Các nút tính toán và các cấu hình mạng của chúng.
- Các hệ thống con và interface của chúng.
- Các class thiết kế có ý nghĩa kiến trúc như các class chủ động.
- Các cơ chế thiết kế tổng quát xử lý các yêu cầu chung như tính bền vững, hiệu quả,... (được nắm bắt trong các class phân tích và các dẫn xuất use-case ở cấp phân tích.

Thiết kế kiến trúc : nhận dạng nút và cấu hình mạng

Cấu hình mạng vật lý sẽ ảnh hưởng đến kiến trúc phần mềm, gồm các khía cạnh :

- các nút nào liên quan, khả năng về bộ nhớ và công suất tính của nút.
- kiểu nối kết và kiểu giao thức nào giữa các nút.
- các tính chất về sự nối kết và giao thức như băng thông, độ sẵn sàng, chất lượng...
- cần khả năng tính dư thừa, chế độ đề kháng lỗi, di cư process, sao lưu dữ liệu,...

Thiết kế kiến trúc : nhận dạng hệ thống con và interface của chúng

Chia công việc thiết kế từ đầu hay khi mô hình thiết kế phát triển thành phức tạp cần được chia nhỏ. Một số hệ thống con được dùng lại từ các project khác :

- nhận dạng các hệ thống con cấp ứng dụng
- nhận dạng các hệ thống con cấp giữa và cấp hệ thống
- định nghĩa sự phụ thuộc giữa các hệ thống con.
- nhận dạng giao tiếp của các hệ thống con.

Thiết kế kiến trúc : nhận dạng các class thiết kế quan trọng về kiến trúc

Cần nhận dạng các class thiết kế quan trọng về mặt kiến trúc để làm tiền đề cho hoạt động thiết kế, các class khác sẽ được nhận dạng trong việc thiết kế use-case.

- nhận dạng các class thiết kế từ các class phân tích tương ứng
- nhận dạng các class chủ động khi chú ý yêu cầu đồng thời trên hệ thống :
 - các yêu cầu về hiệu quả, độ sẵn sàng, throughput của hệ thống
 - sự phân tán của hệ thống trên các nút.
 - các yêu cầu khác như khởi động, kết thúc, tránh deadlock, tránh bảo hòa, cấu hình lại các nút, khả năng nối kết.



Thiết kế kiến trúc : nhận dạng các cơ chế thiết kế tổng quát

Từ các yêu cầu chung và đặc biệt đã được nhận dạng trong phần phân tích (trong các class phân tích và các dẫn xuất use-case cấp phân tích), quyết định cách xử lý chúng dựa trên công nghệ hiện thực và thiết kế sẵn có. Kết quả là 1 tập các cơ chế thiết kế tổng quát. Các yêu cầu cần xử lý thường liên quan đến :

- tính bền vững.
- sự phân tán & đồng thời.
- các tính chất an toàn dữ liệu.
- đề kháng với lỗi.
- quản lý giao tác.

Thiết kế Use-Case

Mục đích của thiết kế use-case là :

- Nhận dạng các class thiết kế và các hệ thống con có object cần cho việc thực hiện 'flow of events-design' của use-case.
- phân tán hành vi của use-case bằng cách cho các object thiết kế và hệ thống con tương tác nhau.
- định nghĩa yêu cầu trên các tác vụ của class thiết kế, hệ thống con và interface của chúng.
- nắm bắt các yêu cầu cấp hiện thực cho use-case.

Thiết kế Use-Case : nhận dạng các class thiết kế

Nhận dạng các class thiết kế như sau :

- nghiên cứu các class phân tích tham gia trong dẫn xuất use-case cấp phân tích tương ứng, nhận dạng các class thiết kế nối với các class phân tích này.
- nghiên cứu các yêu cầu đặc biệt trong dẫn xuất use-case cấp phân tích tương ứng, nhận dạng các class thiết kế hiện thực các yêu cầu đặc biệt này.
- gán nghĩa vụ cho các class thiết kế tìm được.
- nếu còn thiếu 1 vài class cần cho việc thiết kế use-case đặc biệt, kỹ sư use-case nên liên hệ với kiến trúc sư và kỹ sư linh kiện để bàn bạc.
- xây dựng lược đồ class chứa các class thiết kế tìm được.

Thiết kế Use-Case : miêu tả các tương tác giữa các object thiết kế

Dùng lược đồ trình tự :

- lược đồ trình tự chứa các phần tử actor, đối tượng thiết kế và các t.báo giữa chúng.
- nếu use-case có nhiều luồng điều khiển khác nhau, nên tạo lược đồ trình tự cho từng luồng.
- nên chuyển lược đồ cộng tác ở cấp phân tích thành lược đồ trình tự ban đầu, từ đó phát triển thêm chi tiết.
- dùng 'flow of events', duyệt qua các bước trong nó để quyết định các tương tác nào cần thiết giữa các đối tượng thiết kế và phần tử actor.

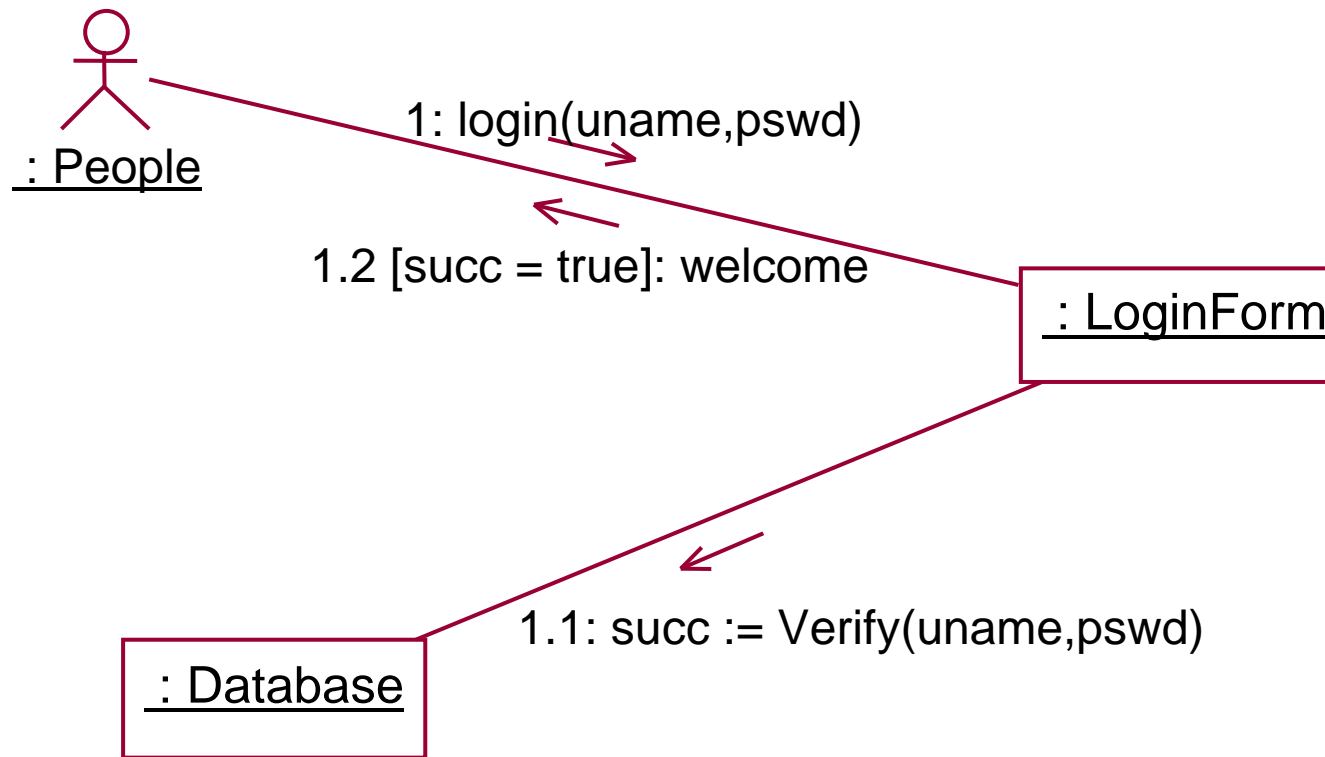
Thiết kế Use-Case : miêu tả các tương tác giữa các object thiết kế

Nên chú ý các ghi nhận sau trong việc xây dựng lược đồ trình tự :

- nên tập trung vào trình tự trong lược đồ.
- actor gửi thông báo đến 1 đối tượng thiết kế để yêu cầu thực hiện use-case.
- mỗi class thiết kế nên có ít nhất 1 đối tượng tham gia vào lược đồ trình tự.
- các message được gửi giữa các đường đời sống đối tượng, có thể có tên tạm và sẽ trở thành tên tác vụ tương ứng.
- dùng label và flow of events cấp thiết kế để bổ sung lược đồ trình tự.
- lược đồ trình tự nên xử lý tất cả mối quan hệ của use-case cần hiện thực.

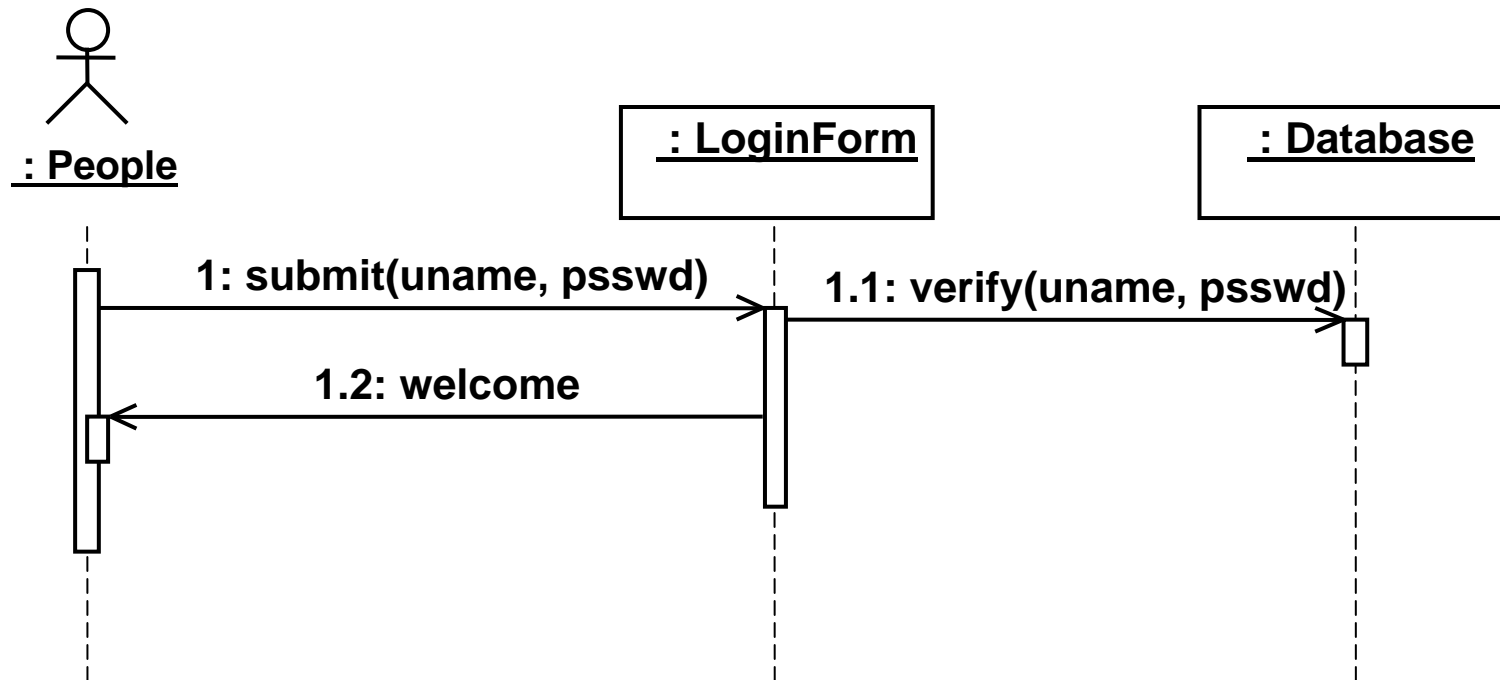
Lược đồ cộng tác

Các thành phần của lược đồ cộng tác : (thí dụ lược đồ cộng tác cho *use-case* Login của hệ thống đăng ký môn học hệ tin chỉ thông qua Web.



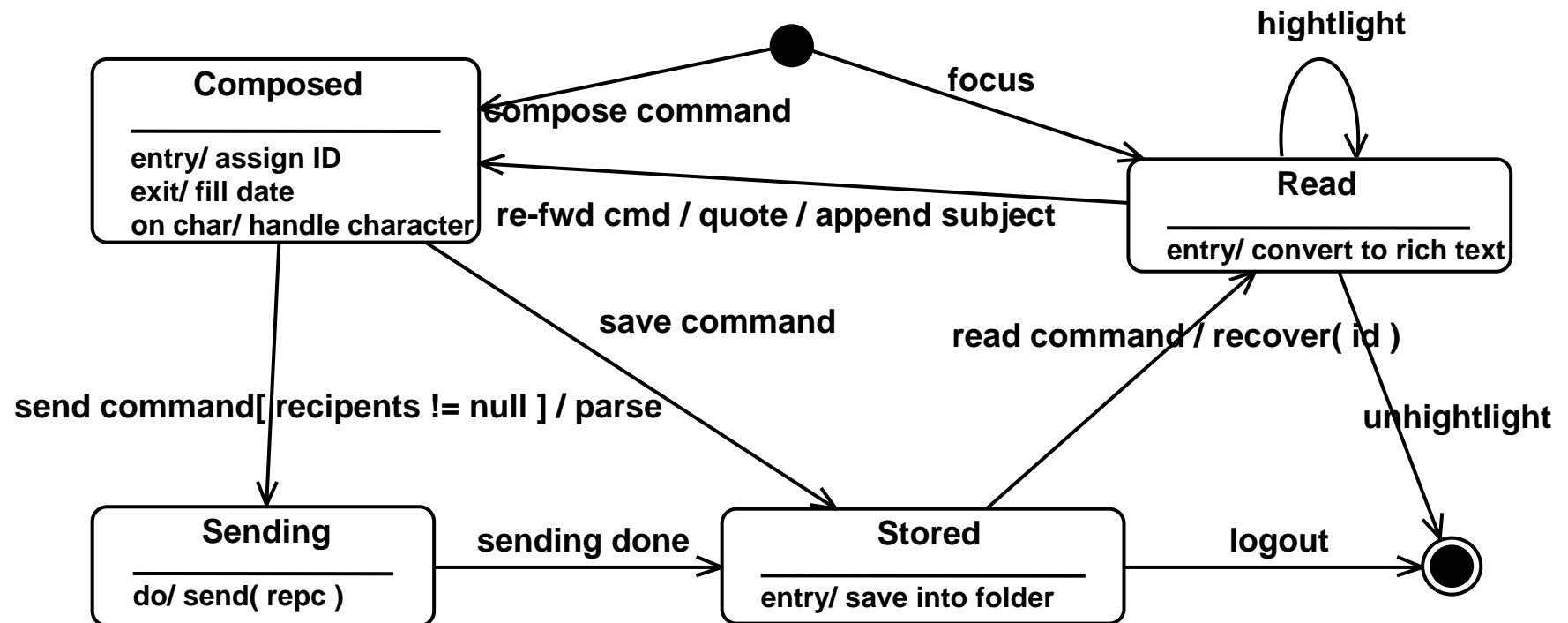
Lược đồ trình tự

Các thành phần của lược đồ trình tự : (thí dụ lược đồ trình tự cho *use-case* Login của hệ thống đăng ký môn học hệ tin chỉ thông qua Web.



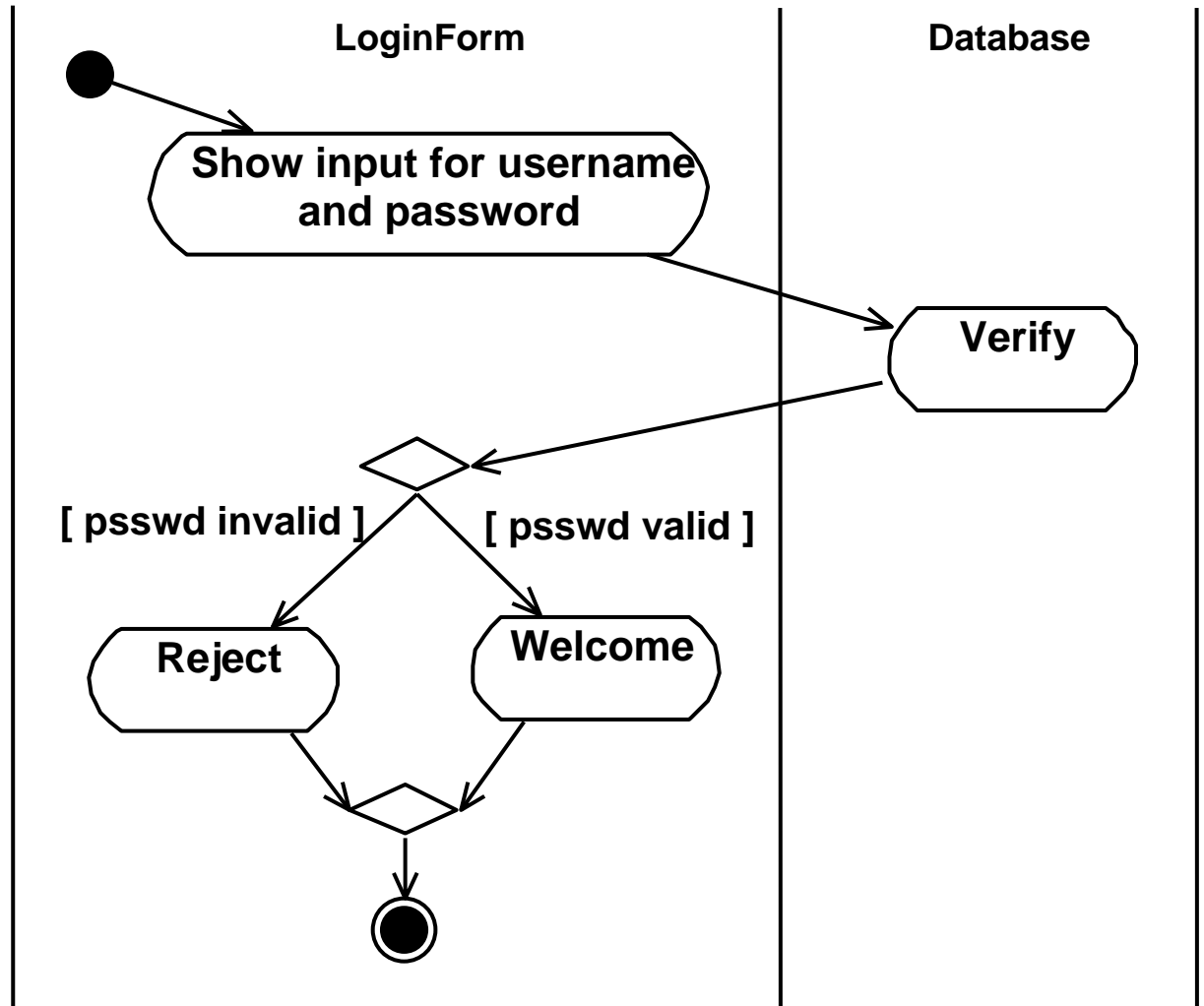
Lược đồ trạng thái

Các thành phần của lược đồ trạng thái : (thí dụ lược đồ trạng thái cho *use-case* Send/Read e-mail).



Lược đồ hoạt động

Các thành phần của lược đồ hoạt động :
(thí dụ lược đồ hoạt động cho use-case Login của hệ thống đăng ký môn học hệ tin chỉ thông qua Web.



Thiết kế class

Mục đích của việc thiết kế class là tạo ra class thiết kế hoàn thành vai trò của nó trong dẫn xuất use-case và các yêu cầu phụ. Bao gồm việc duy trì class thiết kế và các khía cạnh sau của nó :

- các tác vụ của class.
- các thuộc tính của class.
- các mối quan hệ mà class tham gia.
- các method của class (hiện thực tác vụ tương ứng).
- các trạng thái của đối tượng.
- các phụ thuộc tới bất kỳ cơ chế thiết kế tổng quát nào.
- các yêu cầu có liên quan đến hiện thực của class.
- sự hiện thực đúng của bất kỳ interface mà class phải cung cấp.

Thiết kế class : Phát họa class thiết kế

Bước đầu ta phát họa các class thiết kế từ các interface và class phân tích : 1 interface có 1 class thiết kế, còn class phân tích thì :

- thiết kế class biên phụ thuộc vào công nghệ tạo interface : form, activeX control... Để ý dùng các prototype giao diện với user trong bước trước.
- thiết kế class thực thể thường phụ thuộc vào công nghệ database. Việc ánh xạ từ mô hình hướng đối tượng sang mô hình dữ liệu quan hệ có thể cần worker, mô hình và công việc riêng.
- cần tập trung thiết kế class điều khiển, chú ý các nhu cầu sau :
 - vấn đề phân tán : cần nhiều class thiết kế trên các nút khác nhau để hiện thực 1 class điều khiển.
 - vấn đề hiệu quả : nên dùng 1 class thiết kế cho 1 class điều khiển
 - vấn đề giao tác: class thiết kế cần tích hợp công nghệ quản lý giao tác.

Thiết kế class : Nhận dạng các tác vụ

Nhận dạng các tác vụ mà class thiết kế cần cung cấp và tầm vực của chúng (dùng cú pháp ngôn ngữ lập trình). Các đầu vào của bước này là :

- các trách nhiệm của bất kỳ class phân tích mà dẫn tới class thiết kế. 1 trách nhiệm tương ứng 1 hay nhiều tác vụ, nhưng ở đây mới phát họa thông số hình thức các tham số.
- các yêu cầu đặc biệt của bất kỳ class phân tích mà dẫn tới class thiết kế.
- các interface mà class thiết kế phải cung cấp.
- dẫn xuất use-case cấp thiết kế mà class tham gia.

Thiết kế class : Nhận dạng các thuộc tính

Tác vụ thường đòi hỏi thuộc tính, cần dựa vào các hướng dẫn sau :

- chú ý thuộc tính của bất kỳ class phân tích mà dẫn tới class thiết kế. 1 thuộc tính này ám chỉ 1 hay nhiều thuộc tính của class thiết kế.
- hạn chế dùng kiểu của ngôn ngữ lập trình cho thuộc tính.
- cố gắng dùng kiểu đã có.
- nếu class thiết kế quá phức tạp, 1 vài thuộc tính của nó có thể tách ra thành các class riêng.
- nếu có quá nhiều thuộc tính hay thuộc tính quá phức tạp, nên dùng lược đồ class riêng để miêu tả thuộc tính.

Thiết kế class : Nhận dạng các mối quan hệ kết hợp & gộp

Cần theo các hướng dẫn sau :

- chú ý các mối quan hệ kết hợp & gộp của bất kỳ class phân tích mà dẫn tới class thiết kế.
- tính chế số phần tử tham gia, tên vai trò, tính chất của vai trò, class kết hợp, kết hợp n-ary.
- tính chế hướng của mối quan hệ kết hợp từ lược đồ tương tác.

Nhận dạng mối quan hệ tổng quát hóa dùng cú pháp ngôn ngữ lập trình, nếu ngôn ngữ lập trình không hỗ trợ, dùng mối quan hệ kết hợp, gộp để thay thế.

Thiết kế class : Nhận dạng các mối quan hệ kết hợp & gộp

Method đặc tả cách tác vụ được hiện thực. Miêu tả các method dùng ngôn ngữ tự nhiên hay ngôn ngữ pseudocode. Nếu cùng kỹ sư linh kiện thực hiện 2 khâu thiết kế và hiện thực, ông ta thường ít khi đặc tả method trong giai đoạn thiết kế.

Một vài đối tượng thiết kế được điều khiển bởi trạng thái : trạng thái xác định hành vi của nó khi nhận 1 thông báo từ ngoài → miêu tả trạng thái dùng lược đồ trạng thái.

Xử lý các yêu cầu đặc biệt chưa được chú ý ở các bước trước.



Thiết kế hệ thống con (Subsystem)

Mục đích của việc thiết kế hệ thống con là :

- đảm bảo hệ thống con là độc lập với nhau nhiều như có thể có.
- đảm bảo hệ thống con là độc lập với interface của nó nhiều như có thể có.
- đảm bảo hệ thống con cung cấp được interface đúng.
- đảm bảo hệ thống con hoàn thành mục đích, tạo hiện thực đúng cho các tác vụ.



Thiết kế hệ thống con (Subsystem)

Gồm các công việc sau :

- duy trì sự phụ thuộc giữa các hệ thống con (nên thông qua interface), tối thiểu hóa sự phụ thuộc bằng cách bố trí lại các class quá phụ thuộc vào hệ thống con khác.
- duy trì interface của các hệ thống con.
- duy trì nội dung của các hệ thống con.



Chương 8

HIỆN THỰC HƯỚNG ĐỐI TƯỢNG

- Các artifacts cần tạo ra
- Các workers tham gia
- Quy trình hiện thực

Mục đích của giai đoạn hiện thực

Mục đích của hiện thực là :

- kế hoạch các bước tích hợp hệ thống cần thiết cho mỗi bước lập theo cơ chế tăng dần (hệ thống được hiện thực như chuỗi các bước nhỏ và dễ quản lý).
- phân tán hệ thống bằng cách ánh xạ các thành phần khả thi trên các nút trong mô hình bố trí (dựa chủ yếu vào các class chủ động).
- hiện thực các class và hệ thống con thiết kế.
- kiểm tra đơn vị trên các thành phần, tích hợp chúng vào 1 hay nhiều file khả thi trước khi gởi đi kiểm tra tích hợp và kiểm tra hệ thống.

Các artifacts cần tạo ra trong hiện thực

- 📁 Mô hình hiện thực = hệ thống hiện thực :
 - các hệ thống con hiện thực
 - các component (executable, file, library, table, document,...)
 - các interface của hệ thống con và component.
 - kế hoạch tích hợp các "build"
 - đặc tả kiến trúc (view of Implementation model)

Stub, hệ thống con

Stub là 1 thành phần mà phần hiện thực chỉ ở mức độ "template" để phát triển hoặc kiểm tra thành phần khác phụ thuộc và stub.

Stub có thể tối thiểu số thành phần mới cần hiện thực cho mỗi version của hệ thống. có phần hiện thực, nhờ đó làm đơn giản việc tích hợp và kiểm tra tích hợp.

Hệ thống con do cơ chế packaging trong môi trường hiện thực tạo ra :

- package trong java.
- project trong VB.
- thư mục các file trong C++,...

Build

Phần mềm được hiện thực theo từng bước nhỏ để quản lý, mỗi bước ta xây dựng 1 build. Build là 1 version khả thi của hệ thống. Lợi ích của cách tiếp cận này là :

- version khả thi có thể tạo ra khá sớm thay vì phải chờ đợi lâu, việc kiểm tra tích hợp nhờ đó được bắt đầu sớm để có thể demo cho các thành viên nội bộ hay các người liên quan.
- dễ dàng phát hiện điểm yếu và lỗi vì mỗi lần chỉ có 1 phần mới rất nhỏ được thêm vào.
- kiểm tra tích hợp thường nhanh hơn là kiểm tra toàn bộ hệ thống.

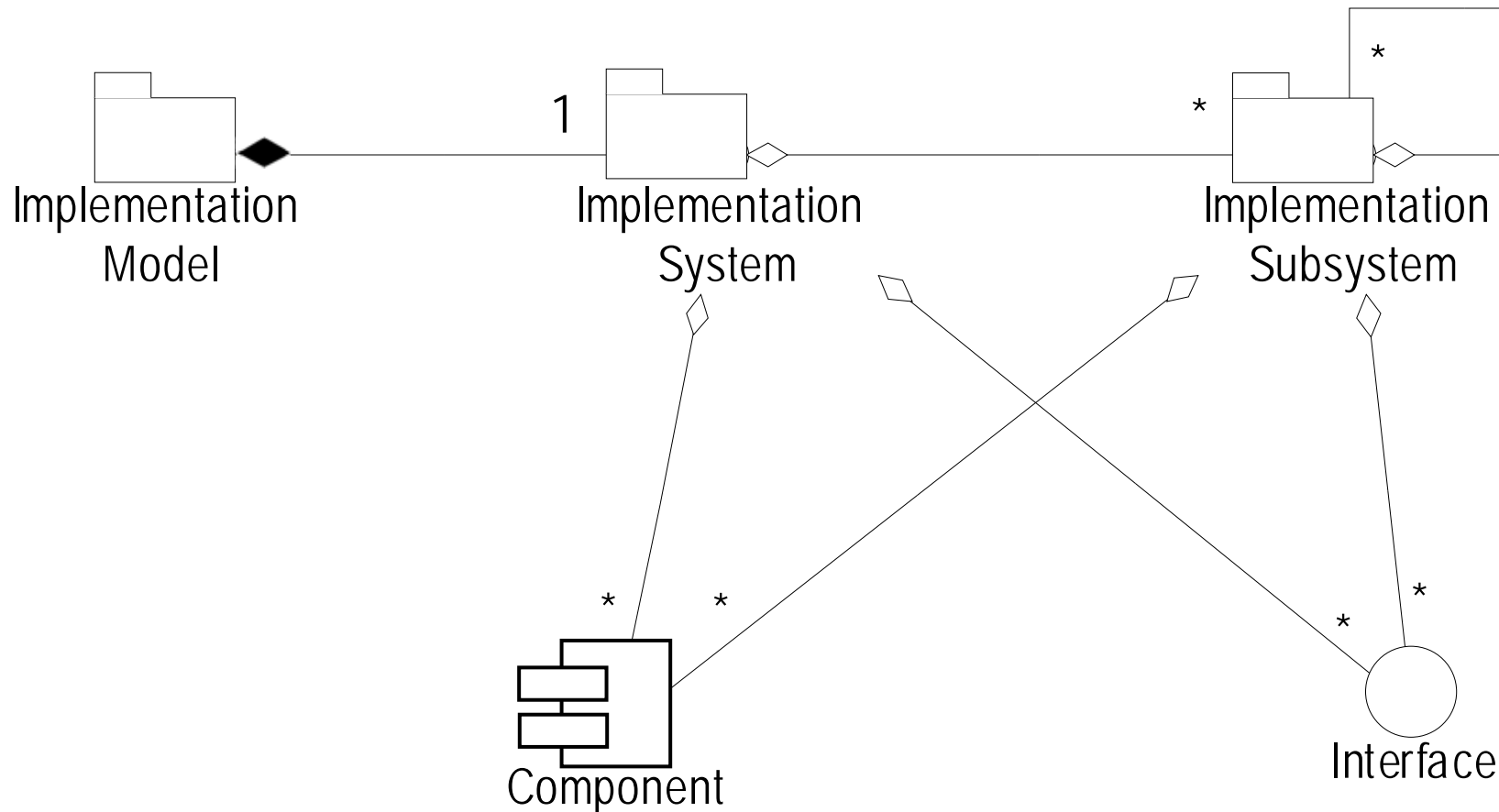
Kế hoạch xây dựng build tích hợp

Kế hoạch xây dựng build tích hợp miêu tả trình tự các build cần thiết cho mỗi bước lặp, ứng với mỗi build nó miêu tả :

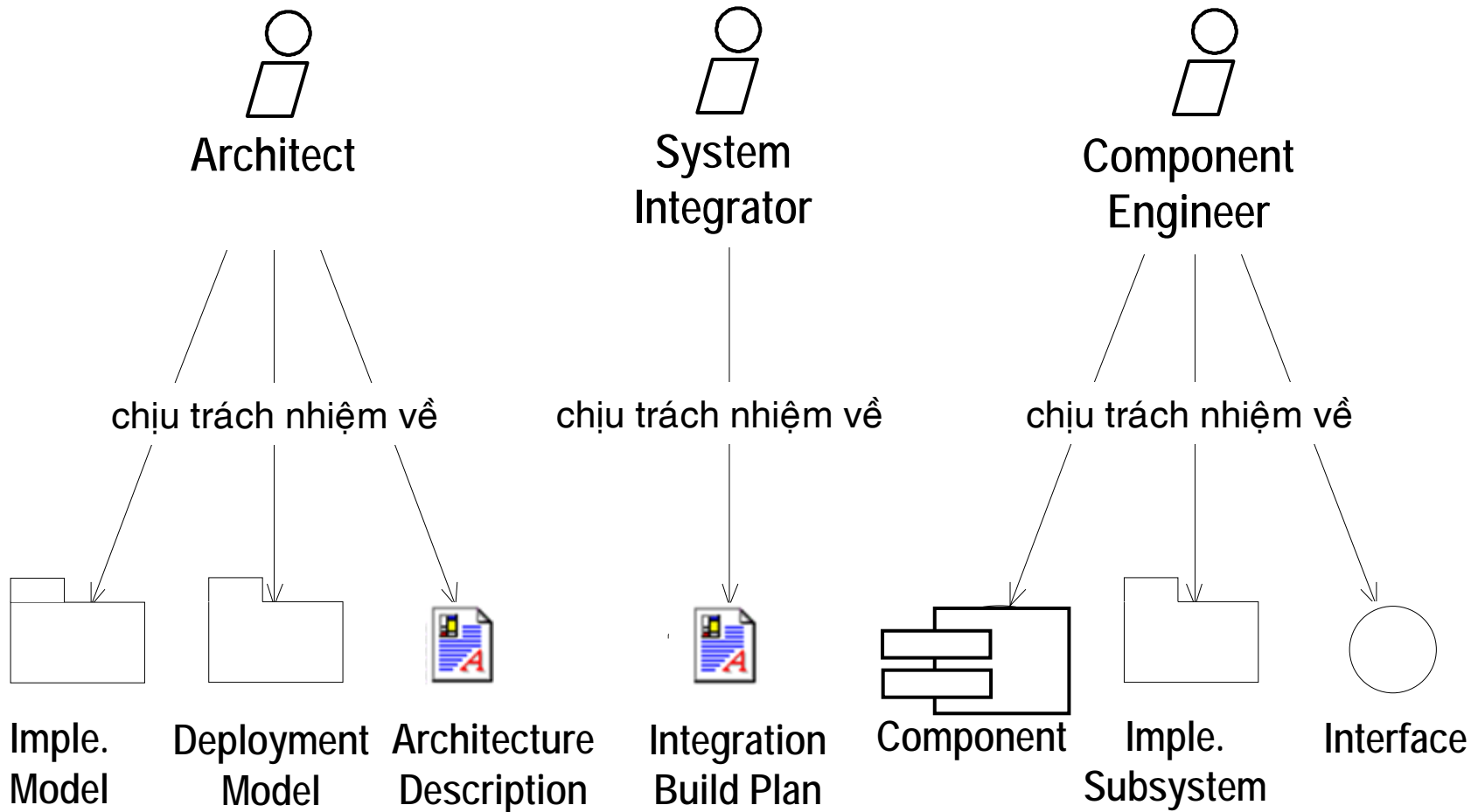
- chức năng kỳ vọng được hiện thực trong build, đây là danh sách các use-case và/hoặc các kịch bản của chúng. Danh sách này cũng chỉ ra các yêu cầu phụ kèm theo.
- Các phần nào của mô hình hiện thực bị tác động trong build, đây là danh sách các hệ thống con và các thành phần được đòi hỏi để hiện thực chức năng kỳ vọng của build.



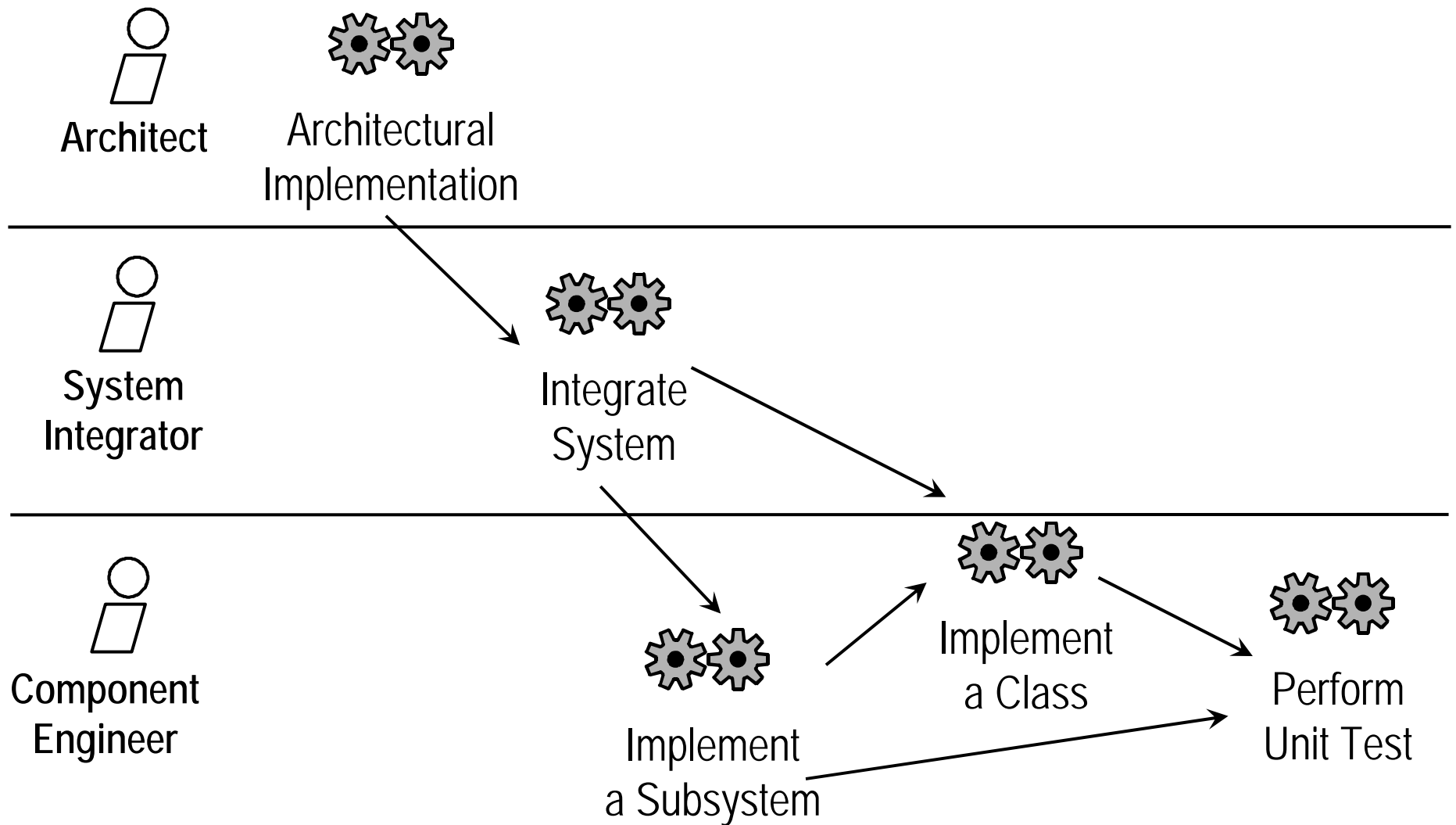
Các artifacts cần tạo ra trong thiết kế



Các workers trong hoạt động hiện thực



Quy trình hiện thực



Hiện thực kiến trúc

Mục đích của hiện thực kiến trúc là phát họa mô hình hiện thực và kiến trúc của nó bằng cách :

- nhận dạng các thành phần có ý nghĩa kiến trúc như các thành phần khả thi (exe).
- ánh xạ các thành phần tới các nút trong cấu hình mạng liên quan : 1 class chủ động được hiện thực trong 1 thành phần khả thi và trở thành 1 process chạy ở 1 nút cụ thể.

Tích hợp hệ thống

Mục đích của tích hợp hệ thống là :

- tạo 1 kế hoạch tích hợp các build miêu tả build nào trong từng bước lập và các yêu cầu trong mỗi build.
- tích hợp mỗi build trước khi kiểm tra tích hợp.

Một vài tiêu chuẩn cho 1 build kế tiếp là :

- build nên thêm 1 số chức năng vào build trước bằng cách hiện thực hoàn chỉnh use-case.
- build không nên bao gồm quá nhiều thành phần mới hay được tinh chế.
- build nên dựa trên build trước theo cơ chế nói rộng từ dưới lên.

Tích hợp hệ thống (tt)

Ứng với mỗi use-case cần hiện thực, làm các việc sau :

- chú ý dẫn xuất use-case ở cấp thiết kế (dẫn ngược về use-case).
- nhận dạng các hệ thống con và class thiết kế trong dẫn xuất use-case cấp thiết kế.
- nhận dạng các hệ thống con hiện thực và các thành phần dẫn ngược về hệ thống con và class thiết kế.
- chú ý ảnh hưởng của việc hiện thực các yêu cầu lên các hệ thống con và các thành phần này, đánh giá ảnh hưởng để quyết định nên hiện thực use-case trong build này hay delay cho build sau.

Tích hợp 1 build

Nếu build được kế hoạch cẩn thận thì việc tích hợp nó rất đơn giản :

- chọn version đúng của các hệ thống hiện thực và các thành phần rồi dịch, liên kết chúng để tạo ra build.
- chú ý mối phụ thuộc dạng bottom-up khi dịch các thành phần và hệ thống con.
- build vừa được tạo ra sẽ được kiểm tra tích hợp và hệ thống nếu nó pass được kiểm tra tích hợp và là build cuối của 1 bước lặp.

Hiện thực hệ thống con

Mục đích của hiện thực hệ thống con là đảm bảo nó hoàn thành vai trò của nó trong mỗi build :

- mỗi class trong hệ thống con thiết kế cần cho build hiện tại nên được hiện thực bởi thành phần trong hệ thống con hiện thực tương ứng (và đệ qui).
- mỗi interface của hệ thống con thiết kế cần cho build hiện tại cũng nên được cung cấp bởi hệ thống con hiện thực tương ứng (và đệ qui)..



Hiện thực class

Mục đích của hiện thực class là hiện thực từng class thiết kế thành file thành phần tương ứng, gồm các công việc sau :

- Phát họa 1 file thành phần chứa source code của class. 1 file có thể chứa nhiều class, nhưng theo qui định của ngôn ngữ lập trình và nguyên lý phân chia module, có thể 1 file chỉ chứa 1 class (VC++, java).
- tạo source code từ class thiết kế và các mối quan hệ mà class tham gia.
- hiện thực các tác vụ của class thiết kế dưới dạng các method.
- đảm bảo thành phần cung cấp cùng interface như class thiết kế.

Khả năng tái sử dụng

Khả năng tái sử dụng phụ thuộc vào các yếu tố sau :

- Giữ cho mỗi method có độ cố kết cao : nên thực hiện 1 chức năng rõ ràng.
- giữ cho mỗi method đủ nhỏ : nếu chiếm nhiều trang thì tách ra nhiều method.
- giữ cho các method thống nhất : nên dùng cùng danh sách tham số cho các method có ý nghĩa sử dụng giống nhau.
- tách biệt method chiến lược và phương thức thực thi.
- mở rộng method càng nhiều càng tốt : khái quát hóa kiểu tham số, số tham số,...
- tránh dùng dữ liệu toàn cục.

Khả năng mở rộng

Khả năng mở rộng phụ thuộc vào các yếu tố sau :

- bao đóng lớp con : ngay cả lớp bao ngoài cũng không thấy bên trong.
- che dấu cấu trúc dữ liệu : không xuất ra bên ngoài.
- tránh đa liên kết giữ nhiều đối tượng : tham khảo cho phép thực hiện chức năng trên đối tượng tương ứng, tránh dùng nó để tiếp tục truy xuất gián tiếp đến các đối tượng khác.
- tránh dùng lệnh switch trên kiểu đối tượng : bản thân đối tượng tự hiểu mình là ai.
- phân biệt tác vụ public và private.

Lập trình ứng dụng lớn

Là xây dựng 1 phần mềm phức tạp cần nhiều nhóm tham gia đồng thời, giao tiếp giữa các thành viên là rất quan trọng :

- Không bắt đầu lập trình nếu chưa hiểu rõ.
- giữ cho các method dễ hiểu : độ kết định cao và nhỏ.
- tạo cho method dễ đọc : tên biến và kiểu có ý nghĩa rõ ràng.
- sử dụng cùng tên cho các giai đoạn khác nhau : phân tích, thiết kế, hiện thực...
- lựa chọn tên kỹ lưỡng : thể hiện đúng vai trò, nghĩa vụ.
- sử dụng các hướng dẫn và qui tắc lập trình : style...
- lập tài liệu cho các class và các method : mục đích, chức năng...
- xuất bản đặc tả hướng dẫn cách sử dụng class.



Thực hiện kiểm tra đơn vị (Unit test)

Mục đích là kiểm tra chức năng của từng đơn vị được hiện thực 1 cách riêng lẻ. Có 2 loại kiểm tra từng đơn vị :

- kiểm tra đặc tả (black-box testing) : kiểm tra hành vi của đơn vị như được thấy từ ngoài.
- kiểm tra cấu trúc (white-box testing) : kiểm tra sự hiện thực bên trong đơn vị.

Cũng còn 1 số loại kiểm tra đơn vị khác như : kiểm tra tính hiệu quả, kiểm tra việc dùng bộ nhớ, kiểm tra tải, kiểm tra khả năng.

Việc kiểm tra tích hợp và kiểm tra hệ thống sẽ được thực hiện trong giai đoạn kiểm tra.

Kiểm tra đặc tả

Mục đích kiểm tra đặc tả (black-box testing) là kiểm tra hành vi của đơn vị như được thấy từ ngoài :

- kiểm tra xem component cho kết quả gì ứng với từng dữ liệu nhập và trạng thái nào đó.
- Số lượng tổ hợp bộ ba {giá trị dữ liệu nhập, trạng thái bắt đầu, giá trị kết quả} thường rất lớn nên việc kiểm tra tất cả tổ hợp này là không khả thi.
- ta xác định các lớp tương đương của "test case" và chỉ kiểm tra trên các lớp tương đương này. Lớp tương đương là tất cả các "test case" làm đơn vị cho hầu như cùng hiệu ứng.

Kiểm tra cấu trúc

Mục đích của kiểm tra cấu trúc (white-box testing) là kiểm tra sự hiện thực bên trong đơn vị :

- đảm bảo mọi hàng lệnh đều được kiểm tra (ít nhất là chạy 1 lần).
- kiểm tra mọi path cần quan tâm : các path chạy thường nhất, các path dễ gây sai, các path ít được biết về giải thuật nhất, các path có rủi ro cao.

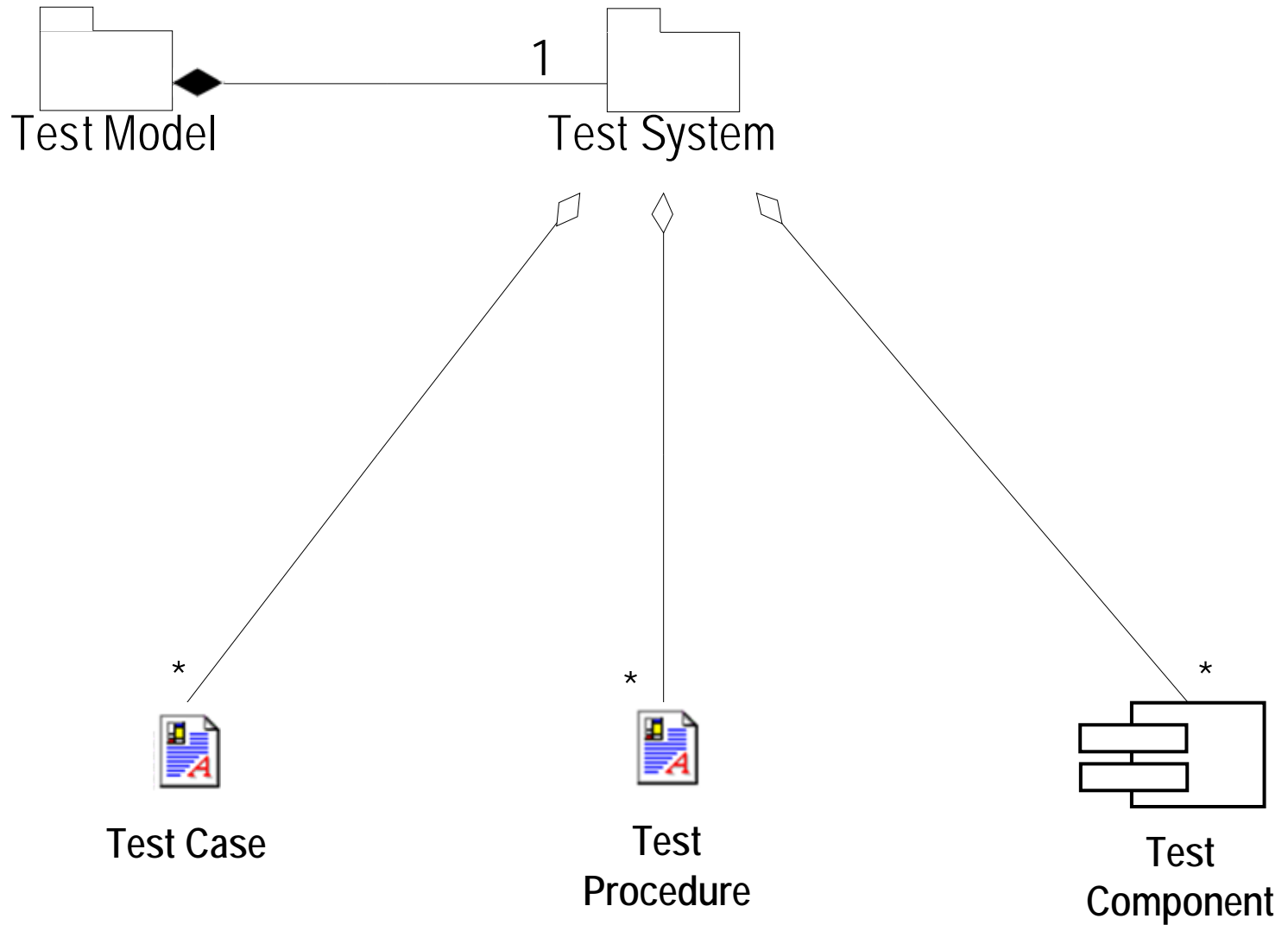
Chương 9
KIỂM THỬ

- Các artifacts cần tạo ra
- Các workers tham gia
- Quy trình kiểm thử

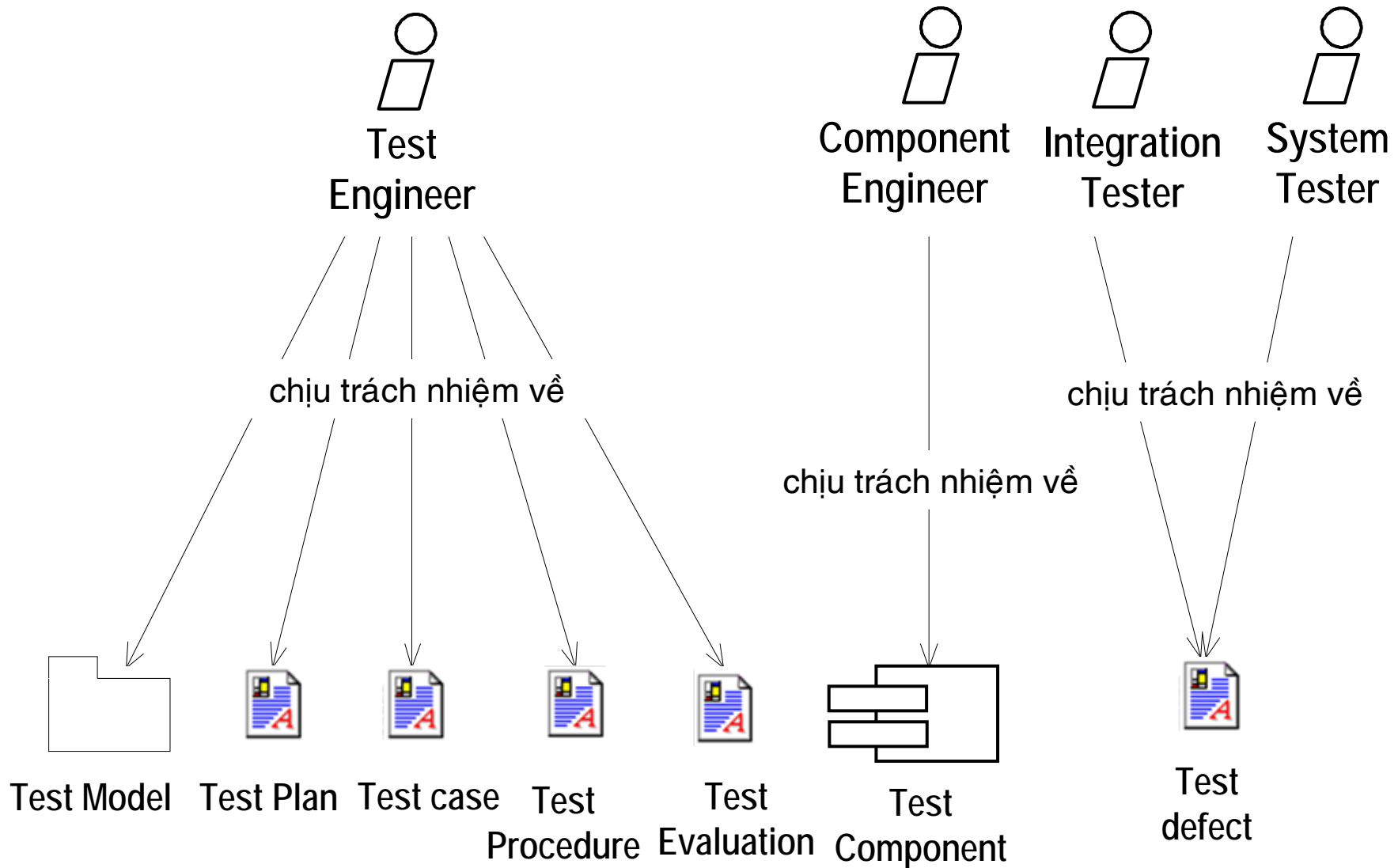
Các artifacts cần tạo ra trong kiểm thử

- 📁 Mô hình kiểm thử = hệ thống kiểm thử :
 - Test case : 1 trường hợp kiểm thử { input, status, output}
 - Test procedure : cách thức thực hiện test case
 - Test component : tự động hóa 1 hay nhiều thủ tục kiểm thử
 - Plan Test : chiến lược, tài nguyên, lịch
 - Defect : lỗi
 - Evaluate Test : phủ các testcase, các code, trạng thái lỗi

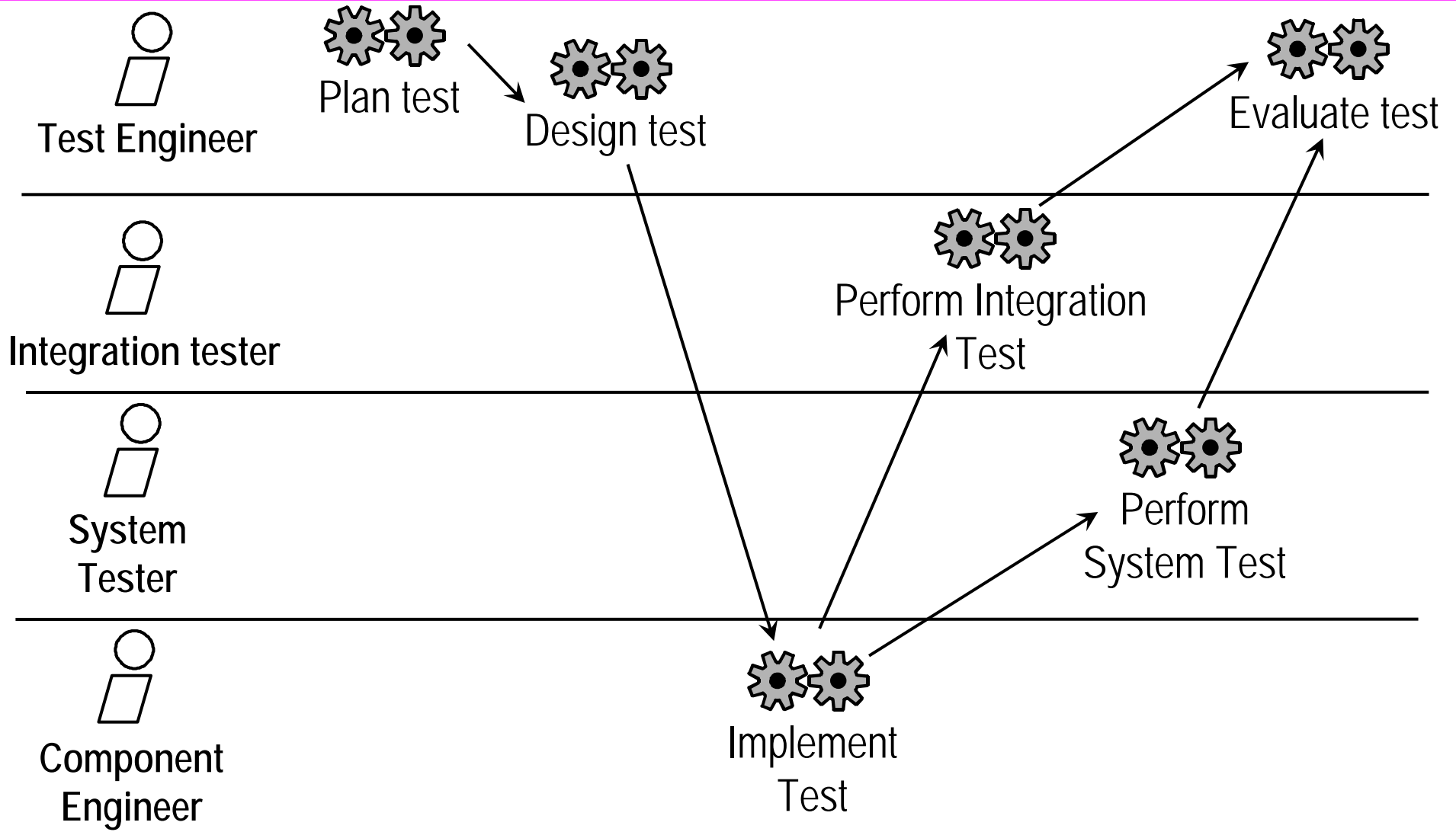
Các artifacts trong kiểm thử



Các workers trong kiểm thử



Quy trình kiểm thử



Kế hoạch việc kiểm thử

Mục đích là kế hoạch hóa các nỗ lực kiểm thử cho 1 bước lặp bằng cách :

- miêu tả chiến lược kiểm thử : thực hiện kiểu kiểm thử nào, thực hiện như thế nào, khi nào thử và đánh giá kết quả kiểm thử như thế nào.
- ước lượng các yêu cầu của nỗ lực kiểm thử, như các tài nguyên về hệ thống hay tài nguyên về người.
- lập lịch cho nỗ lực kiểm thử.

Kỹ sư kiểm thử dùng chủ yếu mô hình use-case và các yêu cầu phụ, cũng có thể dùng thêm mô hình thiết kế.



Thiết kế các test case

Mục đích là :

- nhận dạng và đặc tả các test case cho mỗi build.
- nhận dạng và cấu trúc các thủ tục kiểm thử để xác định cách thực hiện các test case.

Cụ thể :

- thiết kế các test case tích hợp.
- thiết kế các test case hệ thống.
- thiết kế các test case "regression".
- nhận dạng và cấu trúc các thủ tục kiểm thử.



Thiết kế các test case tích hợp

Thiết kế các test case tích hợp :

- test case tích hợp để kiểm tra các thành phần được hiện thực trong build tương tác với nhau, thường dựa vào dẫn xuất use-case ở cấp thiết kế, chủ yếu là các lược đồ tương tác và trình tự.
- để giảm nhẹ nỗ lực thiết kế test case, kỹ sư kiểm thử cố gắng tìm ra các test case có phần giao tối thiểu, mỗi test case tương ứng với 1 path hay 1 kịch bản riêng trong dẫn xuất use-case.
- sau khi kiểm thử, kỹ sư kiểm thử sẽ so sánh các tương tác thực sự giữa các đối tượng với sơ đồ tương tác trong thiết kế, nếu giống thì Ok, nếu khác thì có lỗi.

Thiết kế các test case hệ thống

Các test case hệ thống được dùng để kiểm tra xem các chức năng hệ thống có hoạt động tốt ở mức tổng thể ? Mỗi test case kiểm thử các tổ hợp use-case hoạt động dưới những điều kiện khác nhau như cấu hình phần cứng, mức tải, số actor, kích thước database.

Khi phát triển các test case hệ thống nên để ý độ ưu tiên của các tổ hợp test case mà :

- được đòi hỏi để hoạt động song song.
- có thể được thực hiện song song.
- có ảnh hưởng với nhau nếu được thực hiện song song.
- liên quan tới nhiều process.
- thường dùng tài nguyên hệ thống theo 1 cách phức tạp và không thể tiên đoán.

Thiết kế các thủ tục kiểm thử

- dựa vào từng test case và đề nghị thủ tục kiểm thử cho từng test case.
- cố gắng dùng lại các thủ tục kiểm thử đã có nhiều như có thể (có thể có thay đổi nhỏ).
- 1 thủ tục kiểm thử có thể tác động trên nhiều test case và 1 test case có thể có nhiều thủ tục kiểm thử.

Thực hiện kiểm thử

Mục đích là tự động hóa các thủ tục kiểm thử bằng cách tạo các thành phần kiểm thử (nếu có thể). Dựa vào các thủ tục kiểm thử ta tạo các thành phần kiểm thử theo 2 công nghệ :

- dùng tool tự động kiểm thử.
- viết tường minh thành phần kiểm thử.

Các thành phần kiểm thử thường nhận rất nhiều dữ liệu nhập vào tạo ra nhiều dữ liệu xuất, do đó cần thiết phải hiển thị trực quan được các dữ liệu này để dễ theo dõi (dùng bảng tính hay database).

Thực hiện kiểm thử tích hợp

Thực hiện các test case của mỗi build và ghi nhận kết quả. Quy trình thực hiện :

- thực hiện bằng tay tuân tự các test case hay bằng các thành phần kiểm lỗi tự động.
- So sánh kết quả có được với kết quả kỳ vọng.
- thông báo các lỗi tới kỹ sư linh kiện liên quan.
- thông báo các lỗi tới kỹ sư thiết kế test case để đánh giá kết quả kiểm thử tổng thể.

Thực hiện kiểm thử hệ thống

- mục đích kiểm thử hệ thống là thực hiện các test case được đòi hỏi cho mỗi bước lập và nắm bắt các kết quả kiểm thử.
- kiểm thử hệ thống có thể bắt đầu khi kiểm thử tích hợp cho thấy hệ thống đã thỏa mãn các mục tiêu chất lượng tích hợp (khoảng 95%).
- qui trình kiểm thử hệ thống tương tự như kiểm thử tích hợp.

Các công việc kiểm tra hệ thống :

- Kiểm tra tính phục hồi sau lỗi (Recovery testing)
- Kiểm tra tính bảo mật (Security testing).
- Kiểm tra trong môi trường căng thẳng (Stress Testing).
- Kiểm tra trong hiệu suất (Performance testing).

Đánh giá việc kiểm thử

Kỹ sư kiểm thử quan tâm 2 thước đo chính :

- mức độ hoàn thành kiểm thử.
- độ tin cậy : xu hướng lỗi phát hiện so với xu hướng test case thành công.

Dựa trên xu hướng lỗi, kỹ sư kiểm thử có thể đề nghị các công việc sau :

- thực hiện thêm kiểm lỗi để xác định thêm lỗi.
- nói lỏng tiêu chuẩn kiểm lỗi nếu mục tiêu chất lượng quá cao so với bước lập hiện hành.
- Tách các bộ phận có vẻ đạt độ tin cậy, còn các bộ phận chưa đạt phải được xem xét và kiểm thử lại.

Cuối cùng lập tài liệu về mức độ hoàn thành kiểm thử, độ tin cậy, các công việc đề nghị (ta gọi là đặc tả đánh giá kiểm thử).

Tổng quát về hoạt động debug ứng dụng

- ❑ Sau khi viết code cho ứng dụng xong, ta sẽ thử chạy nó để xác định xem nó giải quyết đúng yêu cầu không. Thường ứng dụng chứa nhiều lỗi sai thuộc 1 trong 3 loại sau :
 1. các lỗi về từ vựng (tên các phần tử, từ danh riêng,..) và cú pháp của các phần tử cấu thành ứng dụng. VC++ sẽ phát hiện các lỗi này triệt để và hiển thị thông báo lỗi cho ta xem xét và sửa chữa. Thường sau khi được VC++ thông báo về các lỗi này, ta dễ dàng sửa chúng.

Ví dụ :

```
ItemRec item;          // sẽ báo sai nếu kiểu ItemRec chưa được định nghĩa
// error C2146: syntax error : missing ';' before identifier 'item'
int a
double b;              // thiếu dấu ; ở trước double
// error C2144: syntax error : missing ';' before type 'double'
If (delta >= 0)         // sai từ vựng vì If phải viết thường
//error C2065: 'If' : undeclared identifier
```

Tổng quát về hoạt động debug ứng dụng

2. các lỗi run-time (giải thuật, tràn bộ nhớ,...). VC++ không thể phát hiện các lỗi này vì chúng thuộc phạm trù ngữ nghĩa. Ứng dụng sẽ chạy theo giải thuật được miêu tả, ta phải tự đánh giá tính đúng/sai về giải thuật, nhưng việc tìm lỗi giải thuật thường rất khó. Để giúp đỡ người lập trình dễ dàng tìm ra các lỗi giải thuật, VC++ cung cấp công cụ cho phép họ kiểm soát được qui trình chạy ứng dụng và truy xuất các biến dữ liệu của chương trình, công cụ này được gọi là 'Debug'.

Ví dụ :

```
double a[50];
```

```
double b;
```

```
...
```

```
b = 3.1416;
```

```
a[50] = 9.0;
```

```
// cũng sẽ làm b = 9.0 vì vùng nhớ của biến b cũng  
// chính là vùng nhớ của phần tử a[50]
```


Tổng quát về hoạt động debug ứng dụng

3. các lỗi do truy xuất tài nguyên đồng thời với các ứng dụng khác (môn HĐH sẽ trình bày cụ thể lỗi này và cách giải quyết). Thí dụ 2 ứng dụng truy xuất tài khoản A đồng thời :

1. hiển thị giao diện & chờ người dùng ra lệnh
2. Người dùng ra lệnh nạp vào tài khoản A số tiền 700USD → xử lý :
 - 21a Đọc tài khoản A vào bộ nhớ,
 - 22a Tăng giá trị tài khoản trong bộ nhớ lên 700USD.
 - 23a Ghi lại giá trị mới.
3. Quay về bước 1

Tài khoản
A

1. hiển thị giao diện & chờ người dùng ra lệnh
2. Người dùng ra lệnh rút tiền từ tài khoản A 500USD → xử lý :
 - 21b Đọc tài khoản A vào bộ nhớ,
 - 22b Giảm giá trị tài khoản trong bộ nhớ đi 500USD.
 - 23b Ghi lại giá trị mới.
3. Quay về bước 1

Nếu tài khoản A là 1000USD và HĐH điều khiển chạy 2 process P1 và P2 theo thứ tự 21a→22a→21b→22b→23b→23a thì kết quả tài khoản A sẽ là 1700USD (giá trị đúng là 1200USD).



Tổng quát về tiện ích debug tích hợp trong VC++

Trong quá trình debug, ứng dụng sẽ ở 1 trong 2 chế độ sau :

- **Pause** : chế độ của ứng dụng trước khi chạy hay khi dừng lại theo 1 điều kiện dừng nào đó của người debug. VC++ sẽ ghi nhớ lệnh sắp thi hành trước khi dừng (lệnh đầu tiên của ứng dụng nếu nó chưa bắt đầu chạy). Do tính lịch sử, ta dùng thuật ngữ PC - program counter để nói về lệnh này. Ở chế độ này, người debug có thể xem trạng thái của ứng dụng : giá trị của các biến dữ liệu để biết ứng dụng chạy đúng hay sai theo yêu cầu, lịch sử gọi hàm trong call strack, thêm/bớt các điều kiện dừng,... điều khiển việc thi hành tiếp theo của ứng dụng, lúc này ứng dụng sẽ chuyển sang chế độ Running.
- **Running** : chế độ mà ứng dụng đang chạy các lệnh của nó đến khi nó gặp 1 điều kiện dừng đã thiết lập trước, lúc này ứng dụng sẽ chuyển về chế độ **Pause**.

Trong quá trình debug, ứng dụng ở chế độ Pause chủ yếu thời gian và người debug tương tác với ứng dụng chủ yếu ở chế độ này. Mỗi khi ứng dụng được chạy tiếp, nó chuyển qua chế độ Running, nhưng sẽ nhanh chóng chạy đến lệnh dừng và chuyển về chế độ Pause (trừ phi bị 'blocked' chờ I/O hay bị 'loop' trong các vòng lặp vô tận).

Conngua - Microsoft Visual C++ [break] - [conngua.cpp]

File Edit View Insert Project Debug Tools Window Help

(Globals) [All global members] TimNuocKe

Debug

← Cửa sổ Debug

im được và ngược lại

```
int x, y;
BOOL RetVal;
RetVal = FALSE;
do { // lặp tìm nước ài kiề tiếp đầ và tìm đầ nước ài kiề cách
while (RetVal==FALSE && Nuocdi[SoNuocdi].huong < 8) {
switch (Nuocdi[SoNuocdi].huong) { //thử hướng ài hiể tải
case 0 :
x = Nuocdi[SoNuocdi].x + 2;
y = Nuocdi[SoNuocdi].y - 1;
break;
case 1 :
x = Nuocdi[SoNuocdi].x + 1;
y = Nuocdi[SoNuocdi].y - 2;
break;
case 2 :
x = Nuocdi[SoNuocdi].x - 1;
y = Nuocdi[SoNuocdi].y - 2;
break;
case 3 :
x = Nuocdi[SoNuocdi].x - 2;
y = Nuocdi[SoNuocdi].y - 1;
break;
case 4 :
x = Nuocdi[SoNuocdi].x - 2;
y = Nuocdi[SoNuocdi].y + 1;
break;
case 5 :
x = Nuocdi[SoNuocdi].x - 1;
y = Nuocdi[SoNuocdi].y + 2;
break;
}
}
} while (RetVal==FALSE);
return RetVal;
```

Cửa sổ Registers

Cửa sổ Memory

Cửa sổ CallStack

Cửa sổ Variable

Cửa sổ Watch

Registers

EAX = 00000000	EBX = 7FFDF000
ECX = 00000000	EDX = 00000080
ESI = 00000000	EDI = 0012FF2C
EIP = 0040130E	ESP = 0012FED0
EBP = 0012FF2C	EFL = 00000293
CS = 001B	DS = 0023
ES = 0023	SS = 0023
FS = 0038	GS = 0000
OV=0	UP=0
EI=1	PL=1
ZR=0	AC=1
PE=0	CY=1

00426AEC = 00000000

Memory

Address: 0x00000000

00000000	?? ?? ?? ?? ?? ?? ??	????????
00000007	?? ?? ?? ?? ?? ?? ??	????????
0000000E	?? ?? ?? ?? ?? ?? ??	????????
00000015	?? ?? ?? ?? ?? ?? ??	????????
0000001C	?? ?? ?? ?? ?? ?? ??	????????
00000023	?? ?? ?? ?? ?? ?? ??	????????
0000002A	?? ?? ?? ?? ?? ?? ??	????????
00000031	?? ?? ?? ?? ?? ?? ??	????????
00000038	?? ?? ?? ?? ?? ?? ??	????????

Call Stack

- TimNuocKe() line 64
- main() line 121 + 5 bytes
- mainCRTStartup() line 206 + 25 bytes
- KERNEL32! 77e7eb69()

Context:

Name	Value
Nuocdi[SoNuocdi]	0
RetVal	0
SoNuocdi	0

Watch

Name	Value
RetVal	0

Các thao tác để xem và hiệu chỉnh biến dữ liệu

Để xem nội dung của 1 biến dữ liệu, người debug có thể :

- dờ chuột đến tên biến trong cửa sổ code, 1 cửa sổ nhỏ chứa giá trị của biến đó sẽ được hiển thị để người debug xem xét.
- Xem nội dung của biến trong cửa sổ “Variable”.
- nhập biểu thức (thường là biến dữ liệu) vào vùng Name của cửa sổ Watch để xem nội dung của nó.

Để hiệu chỉnh giá trị của 1 biến nào đó (do đã bị sai, nhưng muốn sửa lại cho đúng hầu có thể kiểm thử các lệnh còn lại), người debug có thể dờ cursor về cell chứa giá trị hiện hành của biến đó (trong cửa sổ Variable hay trong cửa sổ Watch rồi hiệu chỉnh lại giá trị mới).

Các thao tác để xem vị trí thi hành hiện tại

Để hiển thị cửa sổ chứa danh sách các hàm đang thực hiện dở dang (các hàm lồng nhau theo thứ tự), người debug có thể :

- chọn menu View.Debug Windows.Call Stack.
- ấn phải chuột trên gáy cửa sổ bất kỳ rồi chọn mục “Call Stack”.

Để xem vị trí PC hiện hành (lệnh sắp thực hiện kế tiếp), người debug có thể :

- chọn menu Debug.Show Next Statement (thường khi ứng dụng dừng lại, nó sẽ hiển thị lệnh chạy kế tiếp - lệnh bị dừng với màu tô đặc biệt và có dấu mũi tên ở lề trái của lệnh).



Các lệnh thiết lập điều kiện dừng

Để xem/hiệu chỉnh các điểm dừng (breakpoint), ta chọn menu Edit.Breakpoints để hiển thị cửa sổ Breakpoints, từ đó thực hiện chức năng mong muốn :

- Xem danh sách các điểm dừng hiện hành, có thể xóa hết chúng bằng cách chọn button “Remove All”, có thể xóa từng điểm dừng bằng cách chọn nó rồi ấn button “Remove”.
- Muốn thiết lập điểm dừng mới, chọn tab “Location”, nhập vị trí lệnh cần dừng và điều kiện dừng mong muốn (mặc định là luôn luôn dừng ở vị trí qui định).
- Muốn thiết lập điểm dừng mới dựa trên 1 biến nào đó bị thay đổi giá trị, chọn tab “Data”, nhập biểu thức cần tính toán (biến cần quan tâm).
- Muốn thiết lập điểm dừng mới dựa trên thông báo (message) của Windows, chọn tab “Message”, chọn hàm xử lý, chọn thông báo cần dừng.

Ta có thể (và nên) thiết lập nhiều điểm dừng đồng thời để 'rào chắn' nhiều luồng thi hành khác nhau của chương trình.

Các lệnh điều khiển chạy tiếp ứng dụng

Để chạy tiếp ứng dụng từ vị trí PC hiện hành, người debug có thể :

- o chọn menu Debug.Go để bắt đầu chạy ứng dụng, ứng dụng chỉ dừng lại khi gặp điều kiện dừng nào đó đã được thiết lập.
- o chọn menu Debug.Go để chạy tiếp từ vị trí PC hiện hành, ứng dụng chỉ dừng lại khi gặp điều kiện dừng nào đó đã được thiết lập.
- o chọn menu Debug.Step Over để chạy tiếp 1 lệnh rồi dừng lại (Pause), nếu lệnh thi hành là lệnh gọi thủ tục thì toàn bộ thủ tục sẽ được chạy. Đây là lệnh cho phép thực hiện từng lệnh theo mức vĩ mô.
- o chọn menu Debug.Step Into để chạy tiếp 1 lệnh rồi dừng lại (Pause), nếu lệnh thi hành là lệnh gọi thủ tục thì ứng dụng sẽ dừng lại ở lệnh đầu tiên của thủ tục. Đây là lệnh cho phép thực hiện từng lệnh theo mức vi mô.
- o chọn menu Debug.Step Out để chạy tiếp các lệnh còn lại của thủ tục hiện hành rồi quay về và dừng lại sau lệnh gọi thủ tục này (Pause).
- o chọn menu Debug.Run to Cursor để chạy tiếp ứng dụng từ vị trí PC hiện hành đến lệnh chứa cursor hiện hành rồi dừng lại (Pause).

Các lệnh điều khiển khác

Ngoài ra khi ứng dụng ở trạng thái 'Pause', người debug có thể thực hiện các lệnh sau :

- chọn menu Debug.Stop Debugging để kết thúc việc chạy ứng dụng.
- chọn menu Debug.Restart để kết thúc việc chạy ứng dụng rồi bắt đầu chạy lại từ đầu.

Khi ứng dụng ở trạng thái 'Running', người debug có thể thực hiện các lệnh sau :

- chọn menu Debug.Break để dừng đột ngột việc chạy ứng dụng, lệnh đang thực hiện sẽ được đánh dấu để ta dễ theo dõi. Chức năng này giúp ta biết ứng dụng đang bị 'loop' ở đoạn lệnh nào. Nếu ứng dụng đang bị 'block' chờ biến cố I/O, sẽ không có lệnh nào được đánh dấu cả.

Chương 10

CÁC MẪU CẤU TRÚC

- Mẫu Adapter
- Mẫu Composite
- Mẫu Proxy
- Mẫu Decorator
- Mẫu Flyweight
- Mẫu Facade

Giới thiệu

- ❑ Thiết kế phần mềm là một vấn đề rất khó khăn, nhất là khi phần mềm lớn, mối quan hệ giữa các phần tử nhiều → bản thiết kế thường không hiệu quả hoặc có lỗi.
- ❑ Các lỗi thiết kế thường phải trả giá cao do ảnh hưởng đến nhiều giai đoạn sau (viết code, kiểm tra...).
- ❑ Phương pháp lập trình hướng đối tượng cung cấp cơ chế để có thể xây dựng được phần mềm dễ nâng cấp, thay đổi (VD: đặc tính thừa kế, đa hình...). Tuy nhiên việc xây dựng những phần mềm HĐT như thế phụ thuộc nhiều vào khả năng người thiết kế.
- ❑ Mục tiêu của thiết kế: không chỉ thiết kế những phần mềm đúng mà còn có thể hạn chế hoặc hỗ trợ tái thiết kế trong tương lai.

Giới thiệu (tt)

Có nhiều nguyên nhân dẫn đến tái thiết kế :

- Phụ thuộc vào phần cứng, hệ điều hành (OS) hay phần mềm khác: các phần mềm xác định quá chặt chẽ các thông số phần cứng hay phần mềm liên quan sẽ phải thay đổi khi các thông số này thay đổi.
- Phụ thuộc vào giải thuật: khi hệ thống có nhiều giải pháp, nhiều mức độ xử lý cho cùng một vấn đề, việc ràng buộc chặt chẽ hệ thống với giải pháp cụ thể sẽ dẫn đến khó bổ sung, thay đổi hệ thống.
- Không tổng quát hóa khi lập trình, nhất là lập trình hướng đối tượng. VD: ràng buộc thông số hình thức với đối tượng lớp con thay vì có thể là đối tượng lớp cha.
- Các component liên quan nhau quá chặt chẽ: mối quan hệ giữa các component nhiều dẫn đến hiện tượng thay đổi dây chuyền khi phải thay đổi một component nào đó. VD: lạm dụng thừa kế trong lập trình hướng đối tượng, các component gọi lẫn nhau nhiều...

Giới thiệu (tt)

- ❑ Một biện pháp được đề xuất để có những bản thiết kế tốt: sử dụng lại những mẫu thiết kế của những chuyên gia đã qua kiểm nghiệm thực tế.
- ❑ Mẫu thiết kế (Design pattern) thường có đặc điểm:
 - Là những thiết kế đã được sử dụng và được đánh giá tốt.
 - Giúp giải quyết những vấn đề thiết kế thường gặp.
 - Chú trọng việc giúp cho bản thiết kế có tính uyển chuyển, dễ nâng cấp, thay đổi.



Vai trò của design pattern

- ❑ Cung cấp phương pháp giải quyết những vấn đề thực tế thường gặp đã được đánh giá, kiểm nghiệm.
 - ❑ Là biện pháp tái sử dụng tri thức các chuyên gia phần mềm.
 - ❑ Hình thành kho tri thức, ngữ vựng trong giao tiếp giữa những người làm phần mềm.
 - ❑ Giúp người tìm hiểu nắm vững hơn đặc điểm ngôn ngữ lập trình, nhất là lập trình hướng đối tượng.
- tăng độ tin cậy, tiết kiệm nguồn lực...



Phân loại software patterns

- ❑ Có nhiều loại Software patterns: analysis patterns, design patterns, organization patterns, process patterns... Bài giảng này chỉ tập trung vào Object Oriented Design Patterns (từ đây về sau gọi là Design Patterns hay mẫu thiết kế).
- ❑ Design patterns có ba nhóm chính
 - Structural – Cung cấp cơ chế xử lý những lớp không thể thay đổi (lớp thư viện của third party...), ràng buộc muộn (lower coupling) và cung cấp các cơ chế khác để thừa kế.
 - Creational – Khắc phục các vấn đề khởi tạo đối tượng, hạn chế sự phụ thuộc platform.
 - Behavioral – Che dấu hiện thực của đối tượng, che dấu giải thuật, hỗ trợ việc thay đổi cấu hình đối tượng một cách linh động.

Phân loại Object Oriented Design Patterns

- ❑ Mỗi nhóm Design Pattern có các pattern về lớp (class patterns) và pattern về đối tượng (object patterns).
- ❑ Class patterns dựa trên mối quan hệ thừa kế giữa các lớp, mối quan hệ này là tĩnh (xác định tại thời điểm dịch), do đó class patterns thích hợp cho hệ thống không cần thay đổi động trong thời gian chạy.
- ❑ Object patterns dựa trên mối quan hệ giữa các đối tượng, do đó có thể thay đổi ở thời điểm chạy.

Khả năng ứng dụng design patterns

- ❑ Tìm kiếm đối tượng: việc phân chia hệ thống thành một tập hợp các đối tượng hoạt động hiệu quả là công việc khó khăn. Design pattern giúp đưa ra những đối tượng thường gặp trong những trường hợp thiết kế tương tự đã gặp trước đây.
- ❑ Xác định số lượng và kích thước đối tượng: trong trường hợp hệ thống cần ràng buộc số lượng xác định đối tượng đang hoạt động hay người thiết kế băn khoăn về việc nên tập trung một số chức năng nào đó vào trong 1 đối tượng hay tách ra thành nhiều đối tượng.
- ❑ Xác định interface và hiện thực (implementation) của đối tượng. Hướng chương trình đến đặc điểm: program to an interface, not an implementation.
- ❑ Giúp thiết kế theo hướng tái sử dụng và linh động bằng cách sử dụng mối quan hệ giữa các đối tượng (bao gộp, thừa kế...) một cách phù hợp và thiết kế theo hướng tiên đoán trước các thay đổi trong tương lai.

Cấu trúc design pattern sẽ trình bày

- ❑ Tên
- ❑ Mục tiêu và nhu cầu áp dụng
- ❑ Ví dụ sử dụng
- ❑ Lược đồ class miêu tả mẫu : chứa các phần tử (lớp, đối tượng) trong pattern và mối quan hệ giữa chúng.
- ❑ Các ngữ cảnh nên áp dụng pattern.



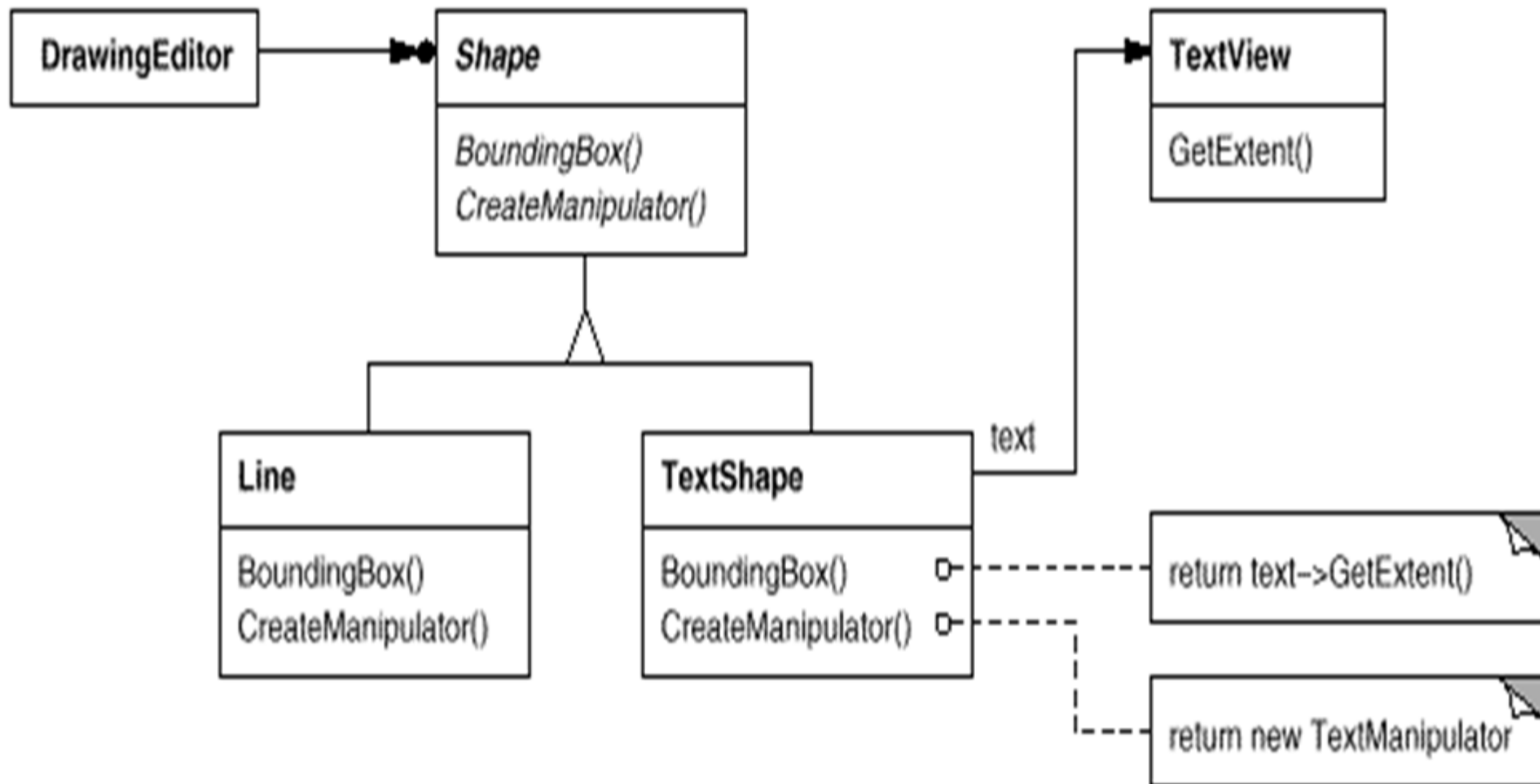
Structural Patterns

- ❑ Các mẫu cấu trúc (Structural Patterns) tập trung giải quyết vấn đề kết hợp các lớp và/hoặc đối tượng thành một kiến trúc lớn hơn.
- ❑ Các mẫu cấu trúc lớp (structural class patterns) sử dụng thừa kế để kết hợp các lớp hay các interface. Tương tự quá trình đa thừa kế: một lớp thừa kế từ nhiều lớp cha sẽ mang đặc điểm của tất cả các lớp cha gộp lại.
- ❑ Các mẫu cấu trúc đối tượng (structural object patterns) tập trung vào việc kết hợp các đối tượng để thực hiện những chức năng nào đó.
- ❑ Trong các slide tiếp theo, chúng ta sẽ tìm hiểu các mẫu: Adapter, Composite, Proxy, Decorator, Facade, Flyweight.

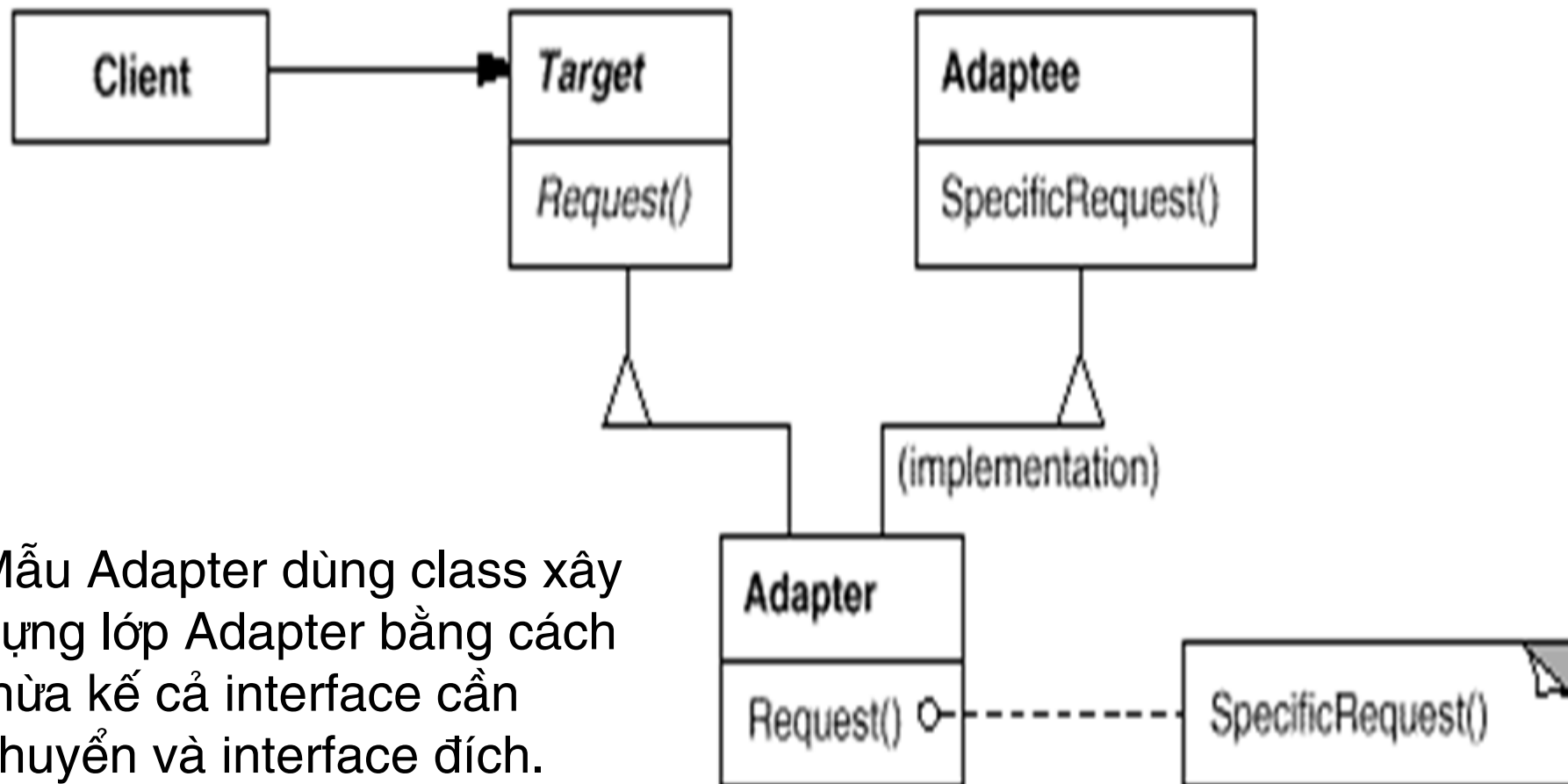
Adapter

- ❑ Mục tiêu : chuyển đổi interface của một class thành một interface khác theo yêu cầu sử dụng của Client.
- ❑ Nhu cầu áp dụng : có những trường hợp chúng ta sử dụng một class nhưng không muốn tuân theo interface của chính class đó mà lại muốn chuyển sang một interface khác. Mẫu Adapter giúp chúng ta giải quyết vấn đề này.
- ❑ Ví dụ : chương trình drawing editor xử lý các đối tượng đồ họa Line, Polygon, Text... thông qua interface sử dụng Shape (được định nghĩa như class root nếu ngôn ngữ lập trình không hỗ trợ Interface). Hiện thực class Line, Polygon từ đầu khá dễ vì đơn giản nhưng hiện thực class Text thì phức tạp hơn → nên dùng lại class sẵn có nào đó (thí dụ TextView cung cấp chức năng quản lý Text) nhưng không thể hay không muốn thay đổi class TextView → định nghĩa class Adapter tên là TextShape thừa kế class Shape của ứng dụng.

Thí dụ về mẫu Adapter dùng object



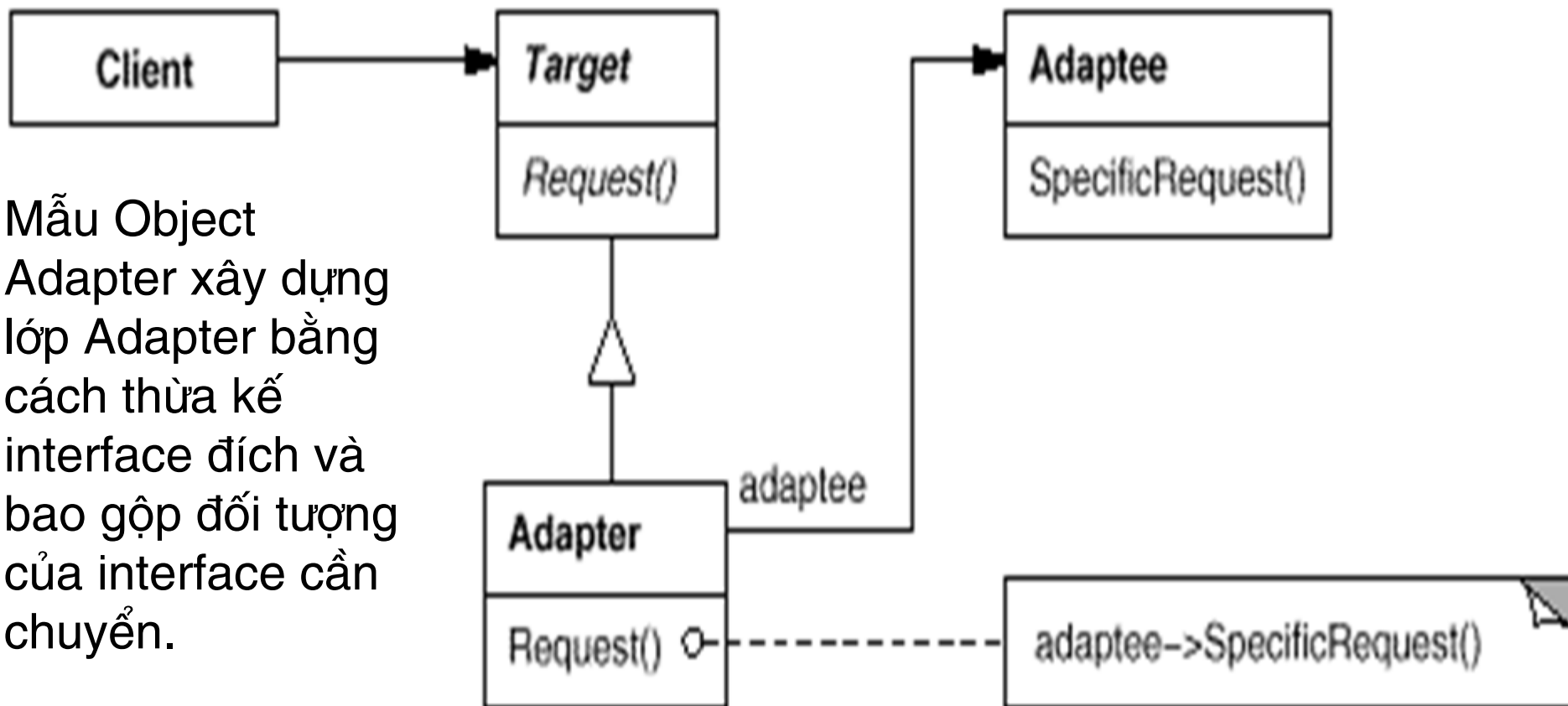
Sơ đồ cấu trúc của mẫu Adapter dùng class



- ❑ Mẫu Adapter dùng class xây dựng lớp Adapter bằng cách thừa kế cả interface cần chuyển và interface đích.

Sơ đồ cấu trúc của mẫu Adapter dùng object

- ❑ Mẫu Object
Adapter xây dựng lớp Adapter bằng cách thừa kế interface đích và bao gộp đối tượng của interface cần chuyển.



Các phần tử tham gia

- ❑ Target (Shape) : định nghĩa interface cho Client sử dụng.
- ❑ Client (DrawingEditor) : sử dụng các đối tượng thông qua interface Target.
- ❑ Adaptee (TextView) : định nghĩa interface đã có sẵn cần “chuyển” sang interface Target.
- ❑ Adapter (TextShape) : “chuyển” interface Adaptee sang interface Target.



Các ngữ cảnh nên dùng mẫu Adapter

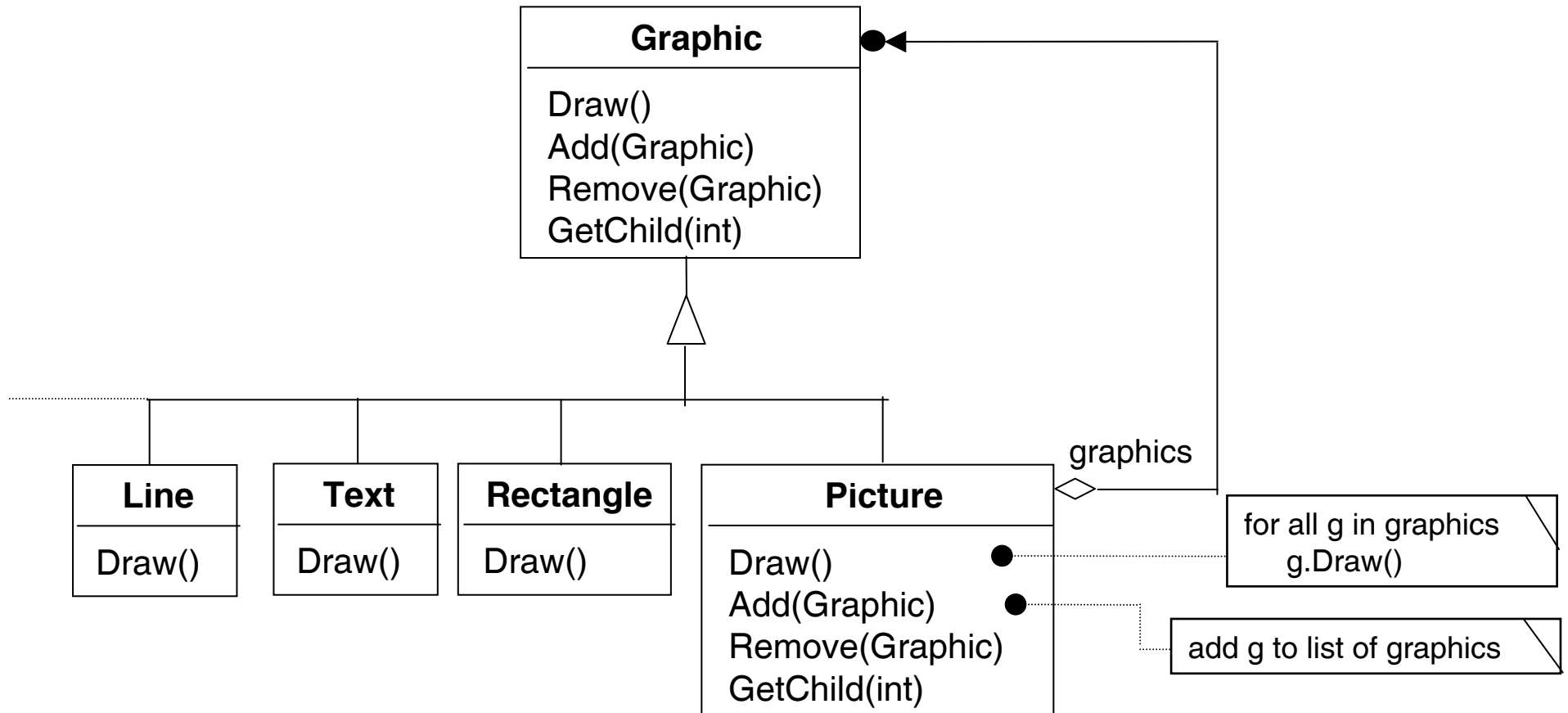
- ❑ muốn dùng một lớp đã có sẵn nhưng interface của nó không tương thích với interface đang sử dụng trong khi chúng ta chỉ muốn dùng interface đang sử dụng.
- ❑ muốn tạo ra các lớp có thể giao tiếp với các lớp khác nhưng chưa biết trước interface của những lớp đó.
- ❑ (đối với mẫu object adapter) muốn sử dụng nhiều lớp con đã có sẵn nhưng sẽ không hiệu quả nếu phải chuyển interface (bằng mẫu Adapter) của từng lớp con. Object Adapter sẽ chuyển interface của chỉ lớp cha.



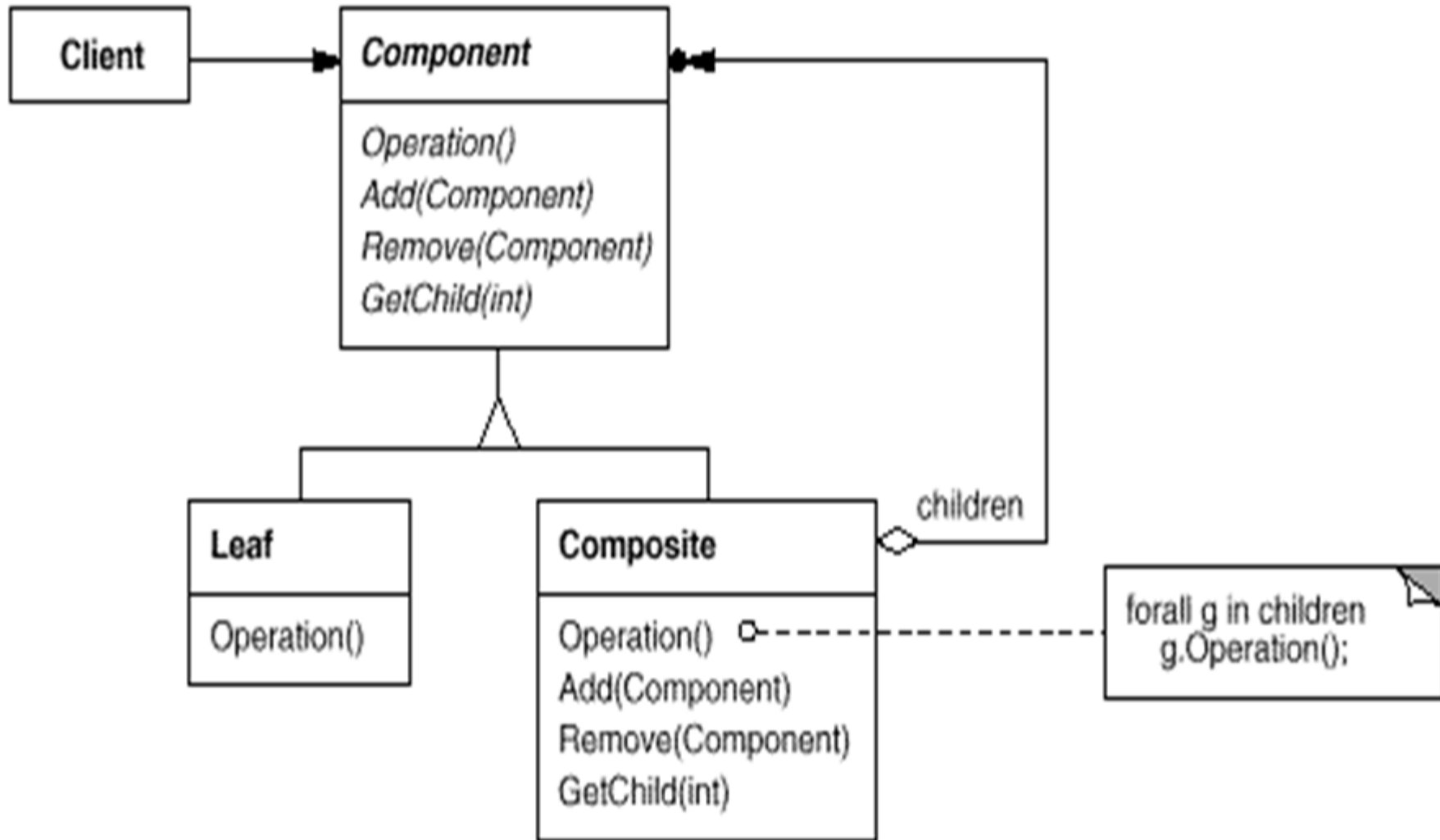
Composite

- ❑ Mục tiêu : tạo quan hệ thứ bậc bao gộp giữa các đối tượng. Client có thể xem đối tượng bao gộp và đối tượng bị bao gộp như nhau → khả năng tổng quát hóa trong code của client → dễ phát triển, nâng cấp, bảo trì.
- ❑ Nhu cầu áp dụng : Có những trường hợp hệ thống muốn xem xét các đối tượng đơn cũng như các đối tượng phức (đối tượng chứa nhiều đối tượng đơn). Trong trường hợp này, hệ thống vừa phải đảm bảo được tính bao gộp lẫn tính không phân biệt giữa các phần tử. Mẫu Composite cung cấp giải pháp cho yêu cầu này.
- ❑ Ví dụ : chương trình drawing editor vừa có các đối tượng đơn như ký tự, điểm ảnh vừa có các đối tượng phức như từ (gồm nhiều ký tự), hàng (gồm nhiều từ), nhóm các phần tử nhỏ hơn... Dưới góc độ người sử dụng, họ thường tác động như nhau lên một từ và một ký tự...

Thí dụ về mẫu Composite



Sơ đồ cấu trúc của mẫu Composite



Các phần tử tham gia

❑ Component

- Khai báo interface và hiện thực một số tác vụ chung cho các đối tượng của những lớp thừa kế (gọi chung là các component)
- Khai báo interface cho việc truy xuất và quản lý đối tượng của các component.
- Có thể khai báo hay hiện thực các phương thức để truy xuất đến đối tượng cha của những component.

❑ Leaf : Định nghĩa tác vụ cho những component cơ bản.

❑ Composite : Định nghĩa tác vụ cho những component bao gộp những component khác.

❑ Client : Sử dụng các component thông qua interface Component



Các ngữ cảnh nên dùng mẫu Composite

- ❑ chương trình muốn thể hiện quan hệ bao gộp - bị bao gộp.
- ❑ chương trình muốn đối xử các phần tử bao gộp và bị bao gộp như nhau.

Ví dụ: compiler (chương trình con hay module có thể bao gộp các chương trình con hay module khác...); chương trình giao diện GUI (window là đối tượng phức, button là đối tượng đơn); chương trình text editor...

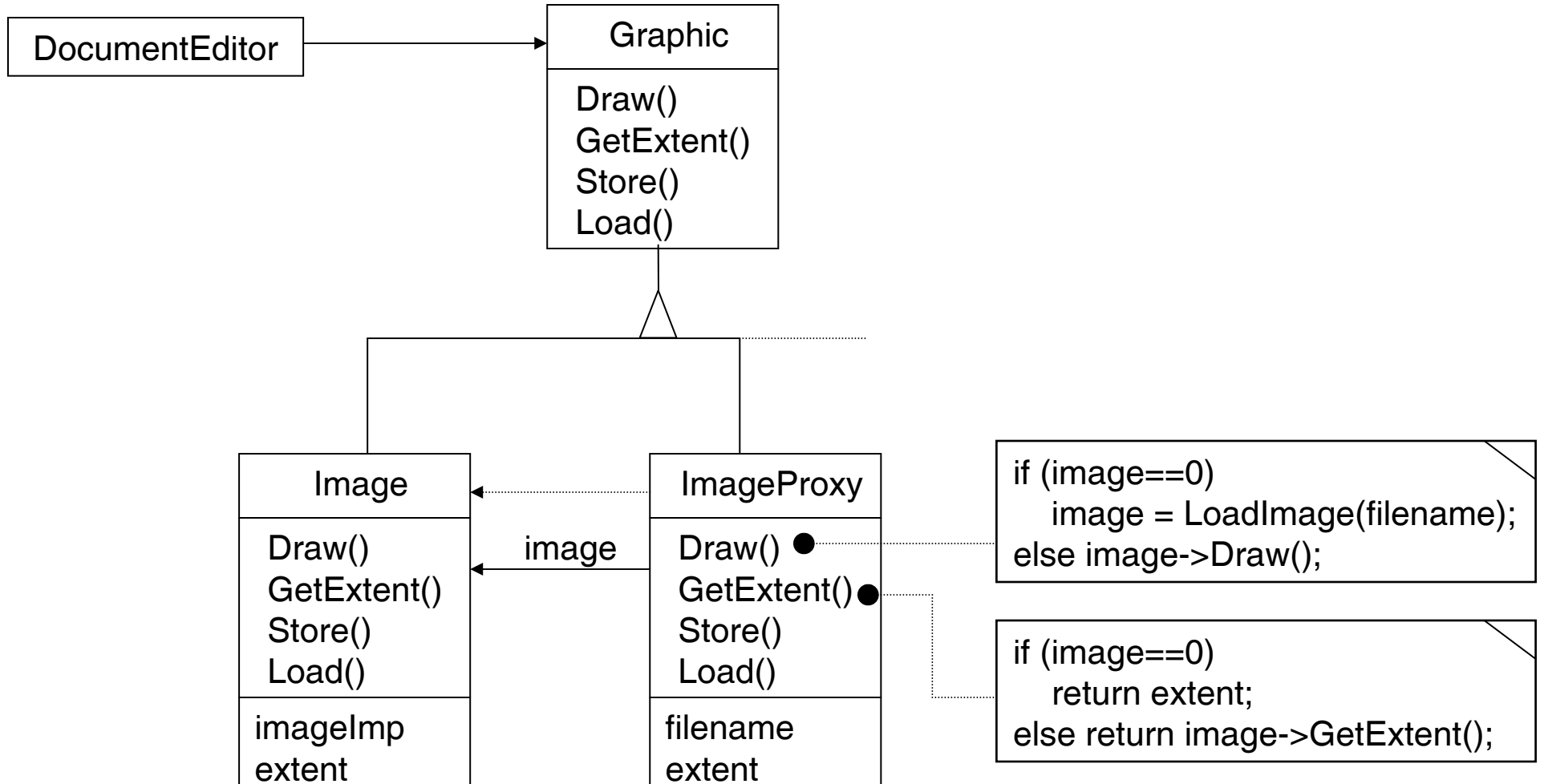
Proxy

- ❑ Mục tiêu : Cung cấp đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó. Đối tượng thay thế gọi là Proxy.
- ❑ Nhu cầu áp dụng :
 - Những đối tượng lớn khi khởi tạo sẽ tốn nhiều tài nguyên, do đó nên trì hoãn thời điểm khởi tạo thực sự các đối tượng này. Trong thời gian trì hoãn, proxy đóng vai trò thay thế đối tượng.
 - Chương trình muốn truy xuất một đối tượng ở không gian địa chỉ khác. Proxy thay thế đối tượng ở máy remote.
 - Đối tượng cần được bảo mật khỏi tương tác trực tiếp với client. Client chỉ tác động được lên Proxy, Proxy chuyển yêu cầu Client xuống đối tượng thực hiện yêu cầu.
 - Chương trình muốn bổ sung một số thao tác kiểm soát lên một đối tượng. Proxy đóng vai trò đối tượng kiểm soát.

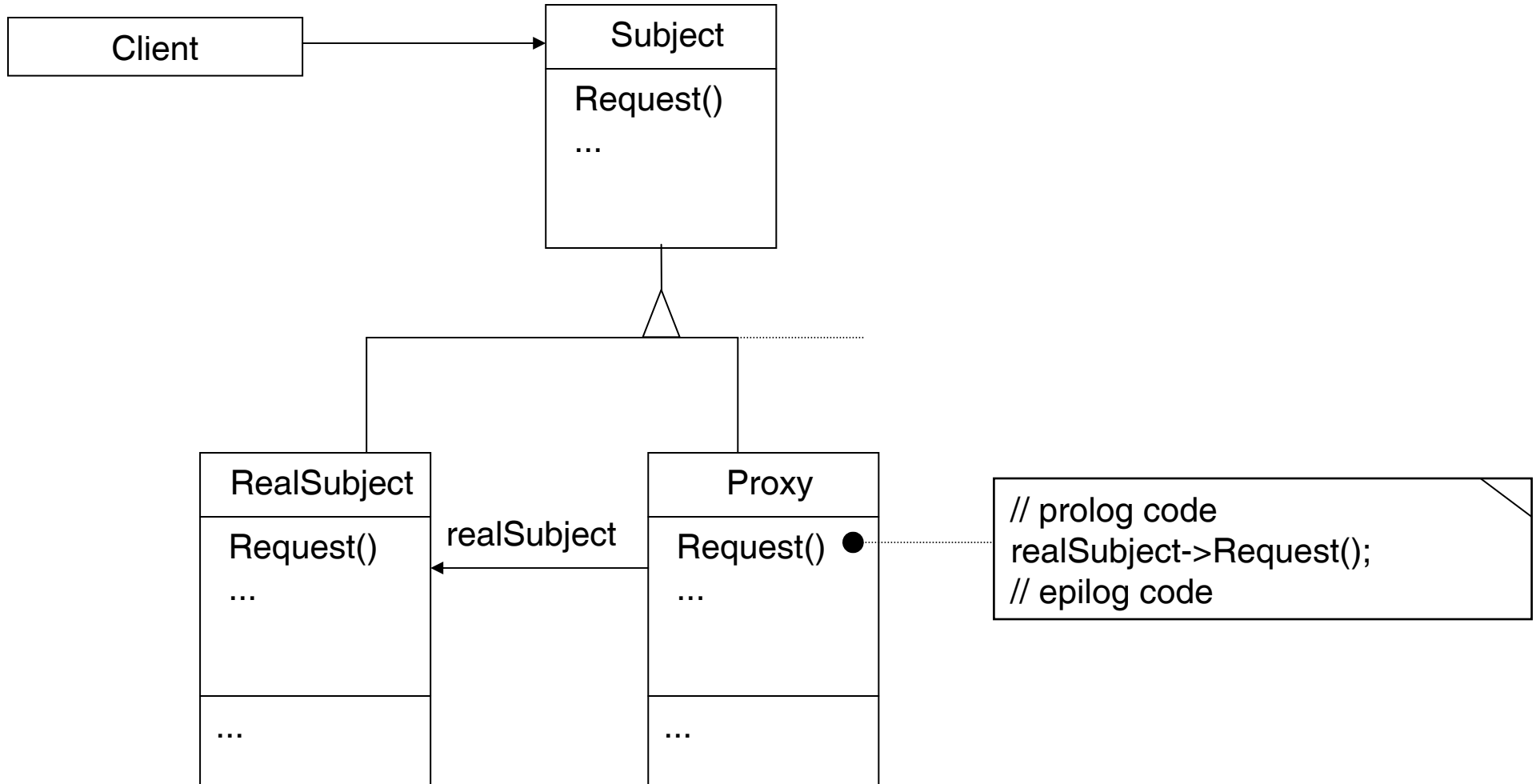
Phân loại Proxy

- ❑ Remote proxy : cung cấp đối tượng đại diện (local) cho một đối tượng phân bố (nonlocal) khác (ví dụ RMI, JINI).
- ❑ Virtual proxy : cung cấp đối tượng đại diện cho đối tượng lớn khi khởi tạo tốn nhiều tài nguyên. Mục đích để trì hoãn thời điểm tạo đối tượng lớn. (Ví dụ đối tượng hình ảnh trong một chương trình xử lý đồng thời nhiều hình ảnh).
- ❑ Protection proxy : cung cấp đối tượng đại diện cho một đối tượng khác cần được bảo mật từ bên ngoài. Ví dụ các KernelProxies cung cấp truy xuất đến Kernel của hệ điều hành.
- ❑ Smart proxy : cung cấp đối tượng đại diện để bổ sung một số thao tác khi có truy xuất đến đối tượng thực. Ví dụ proxy kiểm tra số tham khảo đến đối tượng, proxy thực hiện việc load persistent object trong lần tham khảo đầu tiên...

Thí dụ về mẫu Proxy



Sơ đồ cấu trúc của mẫu Proxy



Các phần tử tham gia

□ Proxy

- giữ liên hệ đến đối tượng RealSubject.
- có thể thay thế đối tượng RealSubject.
- kiểm soát quá trình truy xuất đến đối tượng RealSubject, có thể tạo hoặc delete đối tượng này.
- Thực hiện một số hoạt động khác tùy loại Proxy:
 - + remote proxy: encode và gửi thông tin đến đối tượng RealSubject ở không gian địa chỉ khác.
 - + virtual proxy: chứa các thông tin về đối tượng realSubject để có thể khởi tạo lại nó sau này.
 - + protection proxy: kiểm tra đối tượng đang thực hiện truy xuất có quyền không...
 - + smart proxy: thực hiện các thao tác bổ sung khi có truy xuất đến đối tượng thực.

Các phần tử tham gia

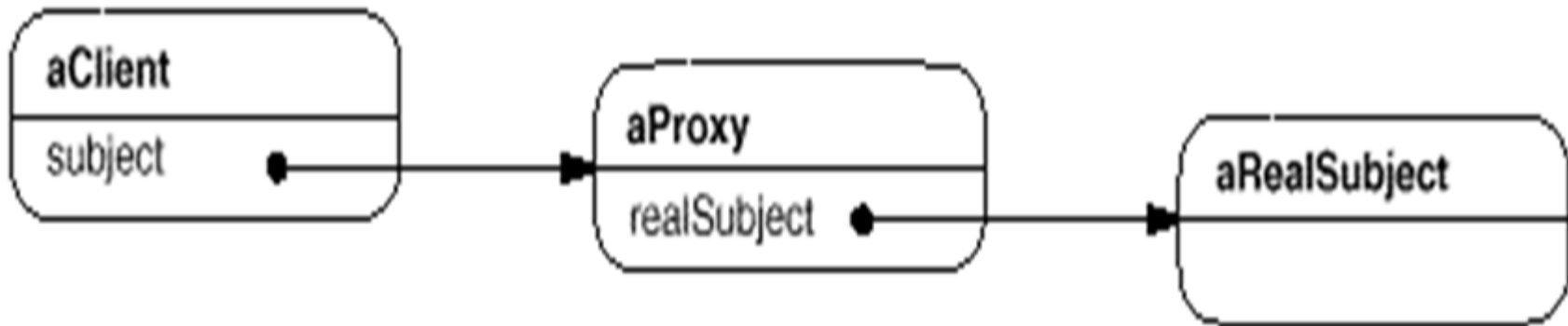
❑ Subject :

Định nghĩa interface chung cho 2 lớp đối tượng RealSubject và Proxy, do đó đối tượng Proxy có thể thay thế vị trí đối tượng RealSubject.

❑ RealSubject:

Lớp thể hiện đối tượng thực sự Client cần truy xuất.

Quá trình giao tiếp ở thời điểm run-time có thể mô tả bằng sơ đồ:



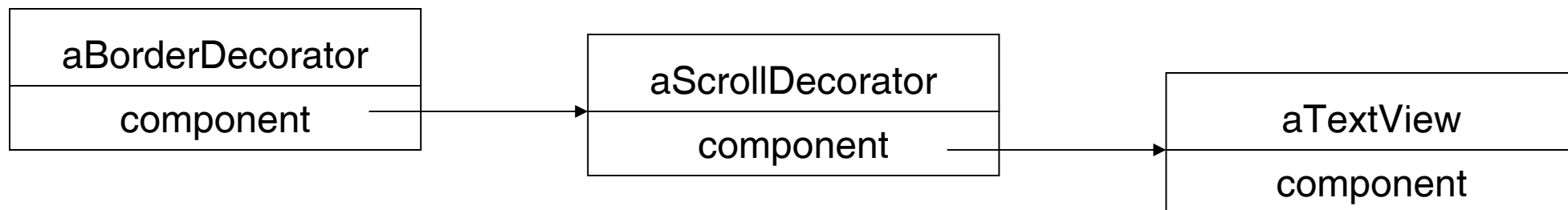
Các ngữ cảnh nên dùng mẫu Proxy

- ❑ Các chương trình phân bố
- ❑ Các hệ thống chương trình cần phối hợp hoạt động của nhiều đối tượng.
- ❑ Các hệ thống middleware.
- ❑ Hệ thống cần chia tải để phục vụ được nhanh.
- ❑ DBMS, OS
- ❑ ...

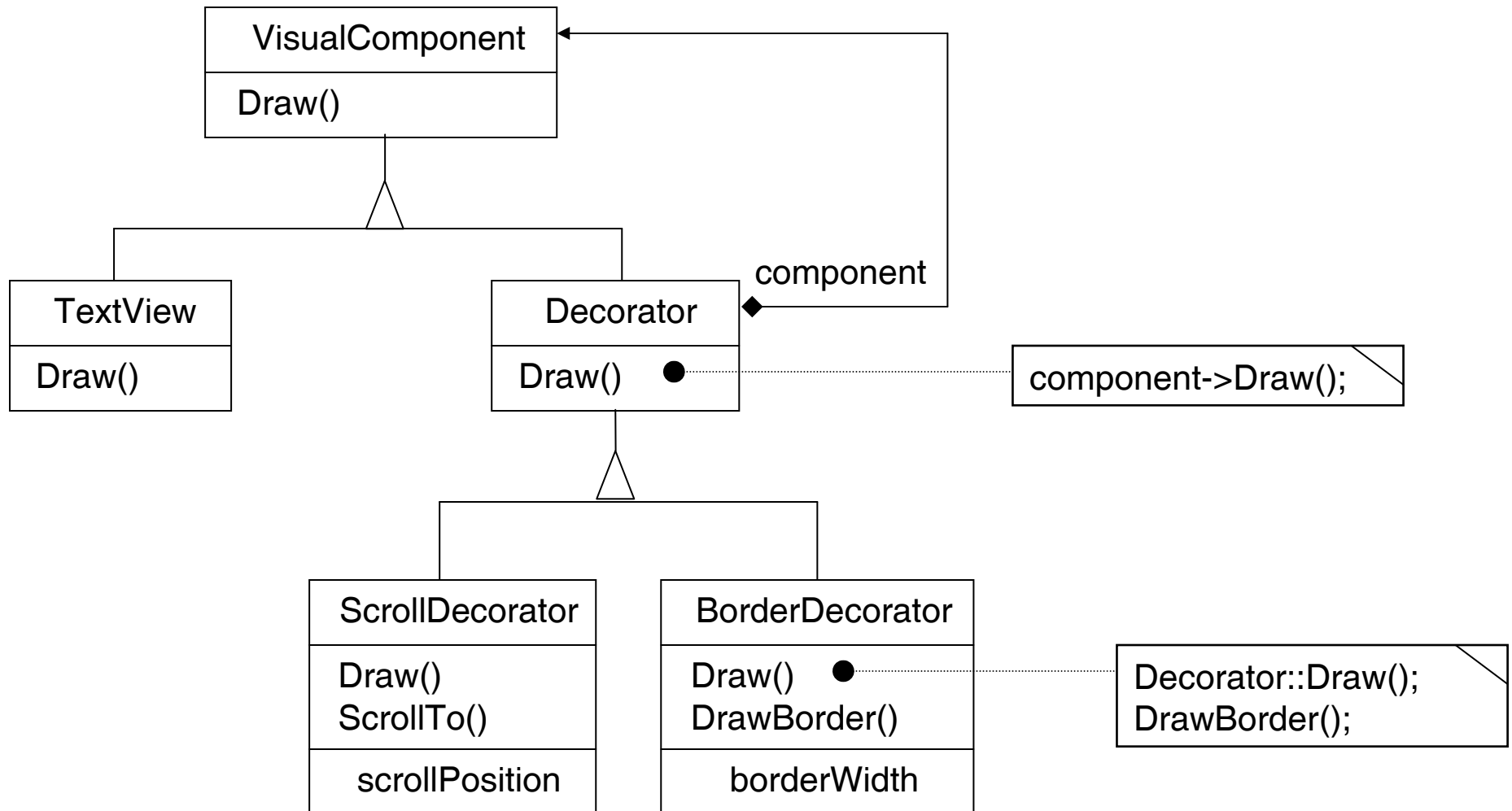


Mẫu Decorator

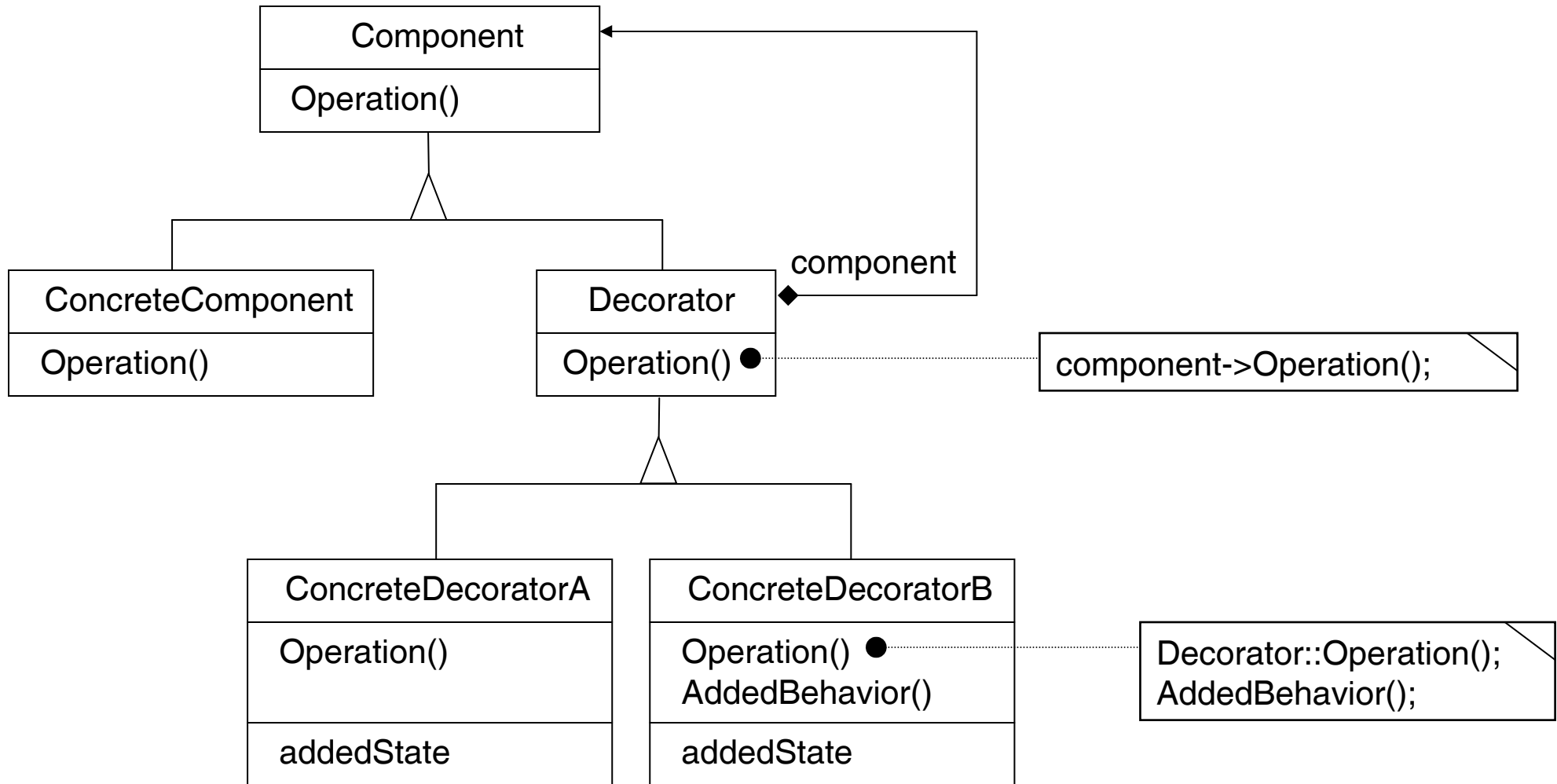
- ❑ Mục tiêu : thêm động trách nhiệm cho đối tượng.
- ❑ Nhu cầu áp dụng :
 - muốn thêm trách nhiệm cho 1 số đối tượng chứ không phải cho toàn bộ các đối tượng của class tương ứng.
- ❑ Ví dụ :
 - Toolkit GUI cho phép user thêm border và scrollbar vào bất kỳ phần tử GUI nào như TextView...



Decorator



Sơ đồ cấu trúc của mẫu Decorator



Các phần tử tham gia

❑ Component (VisualComponent)

- định nghĩa interface cho các đối tượng mà ta cần thêm trách nhiệm cho chúng 1 cách động.

❑ ConcreteComponent (TextView)

- định nghĩa đối tượng mà ta cần thêm trách nhiệm cho chúng 1 cách động.

❑ Decorator

- chứa tham khảo đến đối tượng Component và định nghĩa interface tương thích với interface của Component.

❑ ConcreteDecorator (BorderDecorator, ScrollDecorator)

- thêm trách nhiệm cho thành phần gốc.

Các ngữ cảnh nên dùng mẫu Decorator

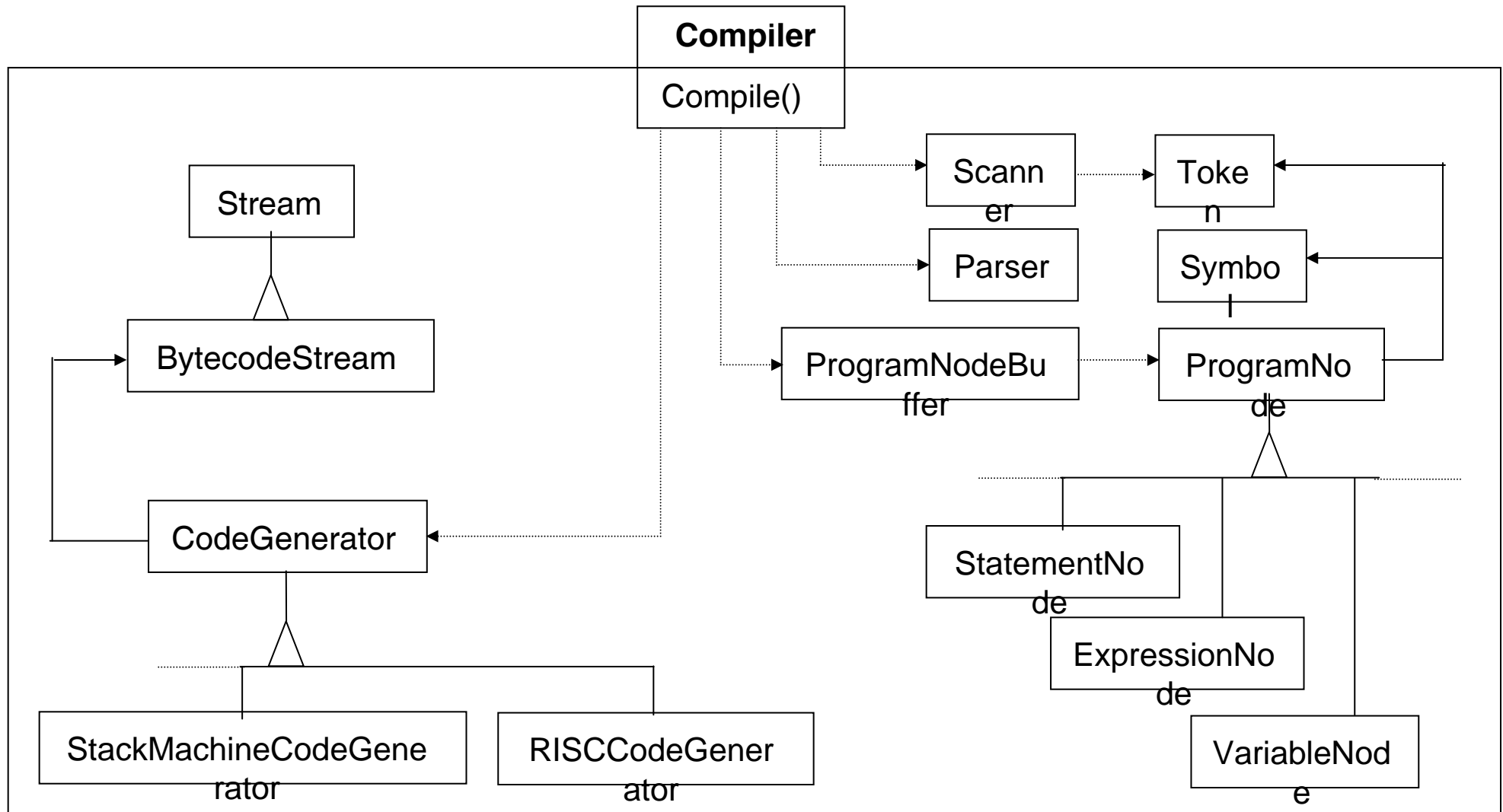
- ❑ muốn thêm động trách nhiệm cho 1 vài đối tượng mà không ảnh hưởng đến các đối tượng cùng loại.
- ❑ tích lũy thêm các trách nhiệm của đối tượng.
- ❑ khi không thể nói rộng đối tượng bằng cách thừa kế (sợ bùng nổ hệ thống class con-cha).



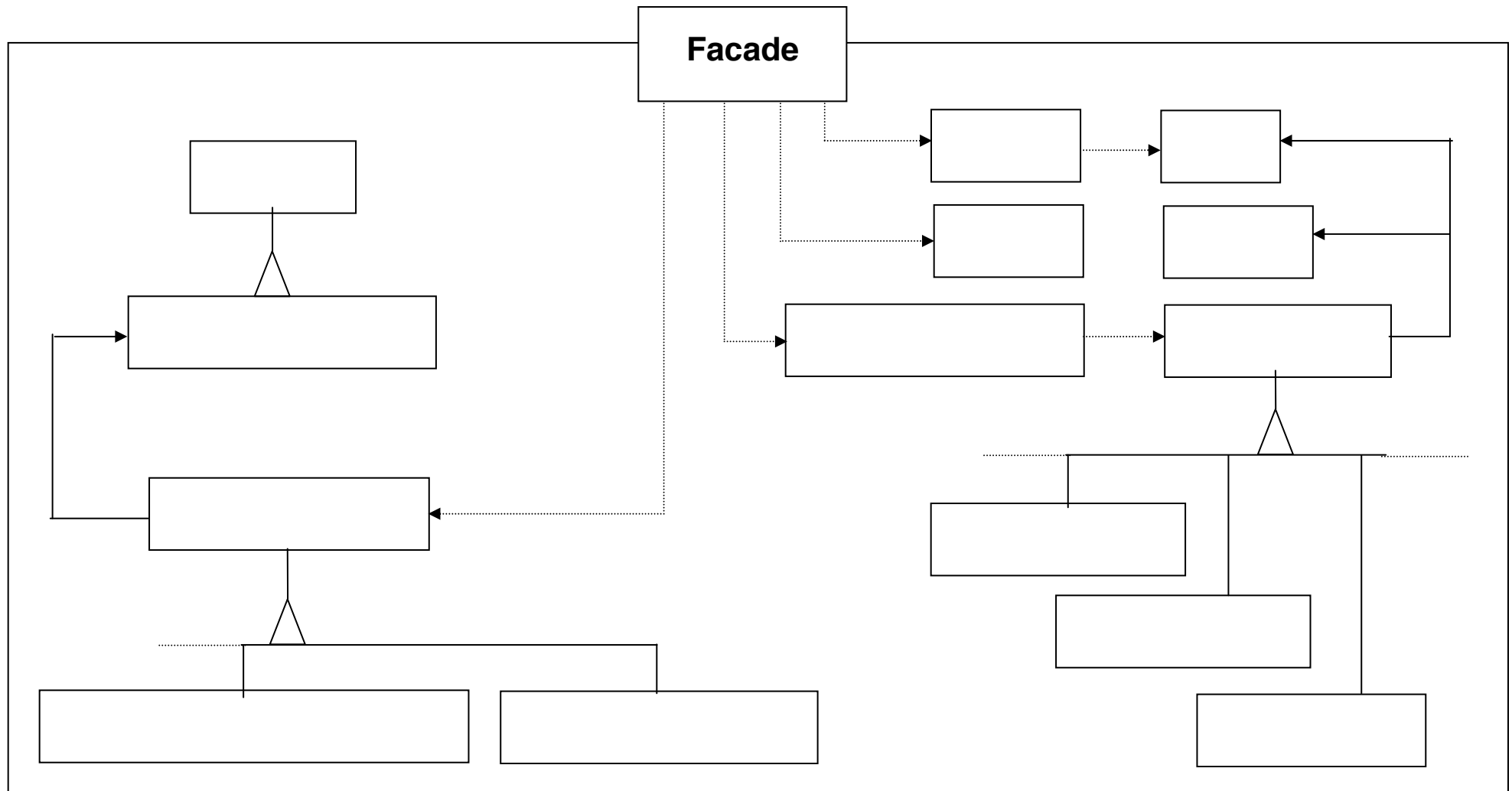
Facade

- ❑ Mục tiêu : cung cấp interface hợp nhất cho tập các interface của 1 hệ thống con. Facade định nghĩa 1 interface cấp cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn.
- ❑ Nhu cầu áp dụng :
 - tối thiểu hóa tính "coupling" giữa các hệ thống con để tối thiểu hóa giao tiếp giữa các hệ thống con.
- ❑ Ví dụ :
 - hệ thống con biên dịch có nhiều class phục vụ các bước biên dịch rời rạc như Scanner, Parser, ProgramNode, BytecodeStream, ProgramNodeBuilder. Để dịch source code, ta có thể viết 1 ứng dụng gọi dịch vụ của từng class để duyệt token, parser, xây dựng cây cú pháp, tạo code đối tượng... Tuy nhiên làm như trên sẽ rất khó và dễ gây ra lỗi. Cách khắc phục là định nghĩa 1 class mới với giao tiếp hợp nhất tên là Compiler, nó cung cấp 1 hàm Compile (file), ứng dụng nào cần dịch source code chỉ cần gọi thông điệp Compile đến đối tượng Compiler.

Thí dụ về cấu trúc Facade



Sơ đồ cấu trúc của mẫu Facade



Các phần tử tham gia

❑ Facade (Compiler)

- biết class nào liên quan đến request xác định.
- nhờ các đối tượng liên quan thực hiện request.

❑ subsystem classes (Scanner, Parser,..)

- hiện thực các chức năng của hệ thống con.
- xử lý công việc được nhờ từ đối tượng Facade.
- không cần biết Facade, không có tham khảo đến Facade.



Các ngữ cảnh nên dùng mẫu Facade

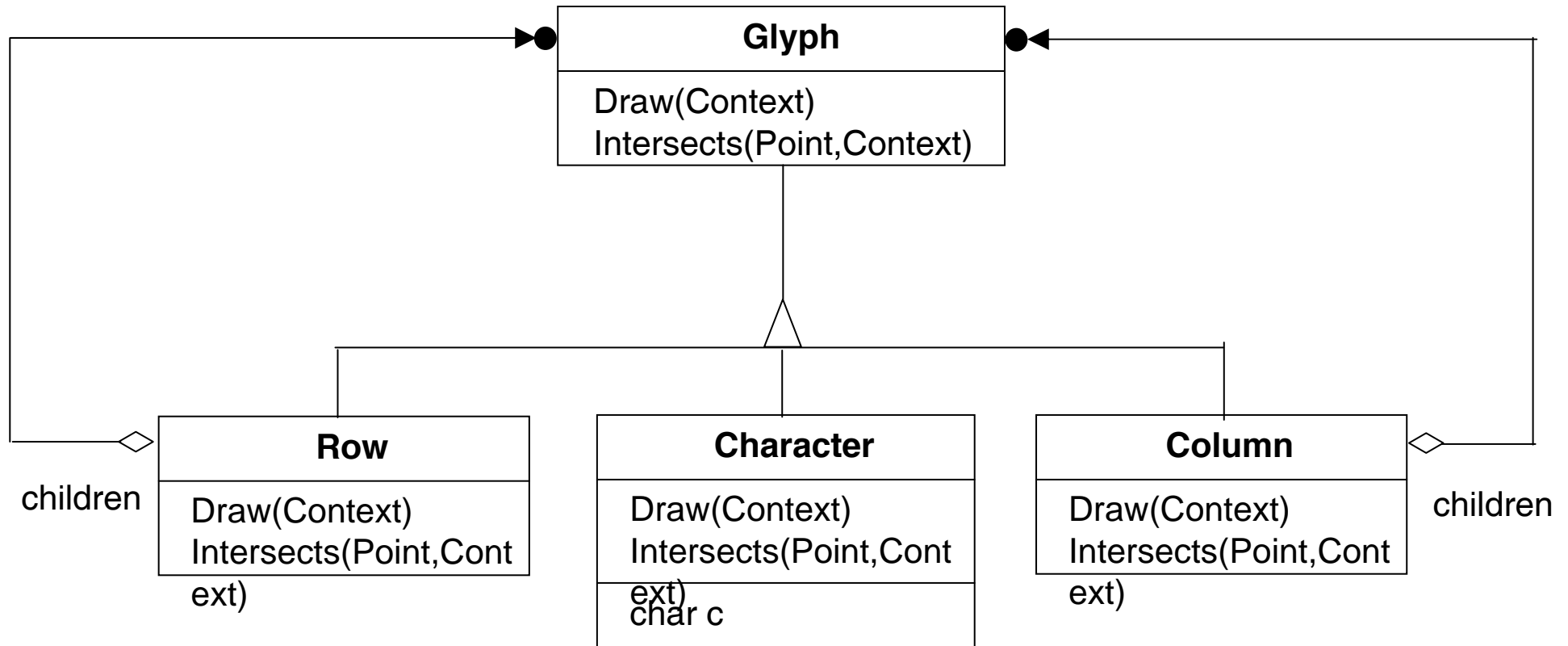
- ❑ muốn cung cấp 1 giao tiếp đơn giản cho 1 hệ thống con phức tạp.
- ❑ muốn tạo thêm lớp ngăn cách hệ thống con với client.



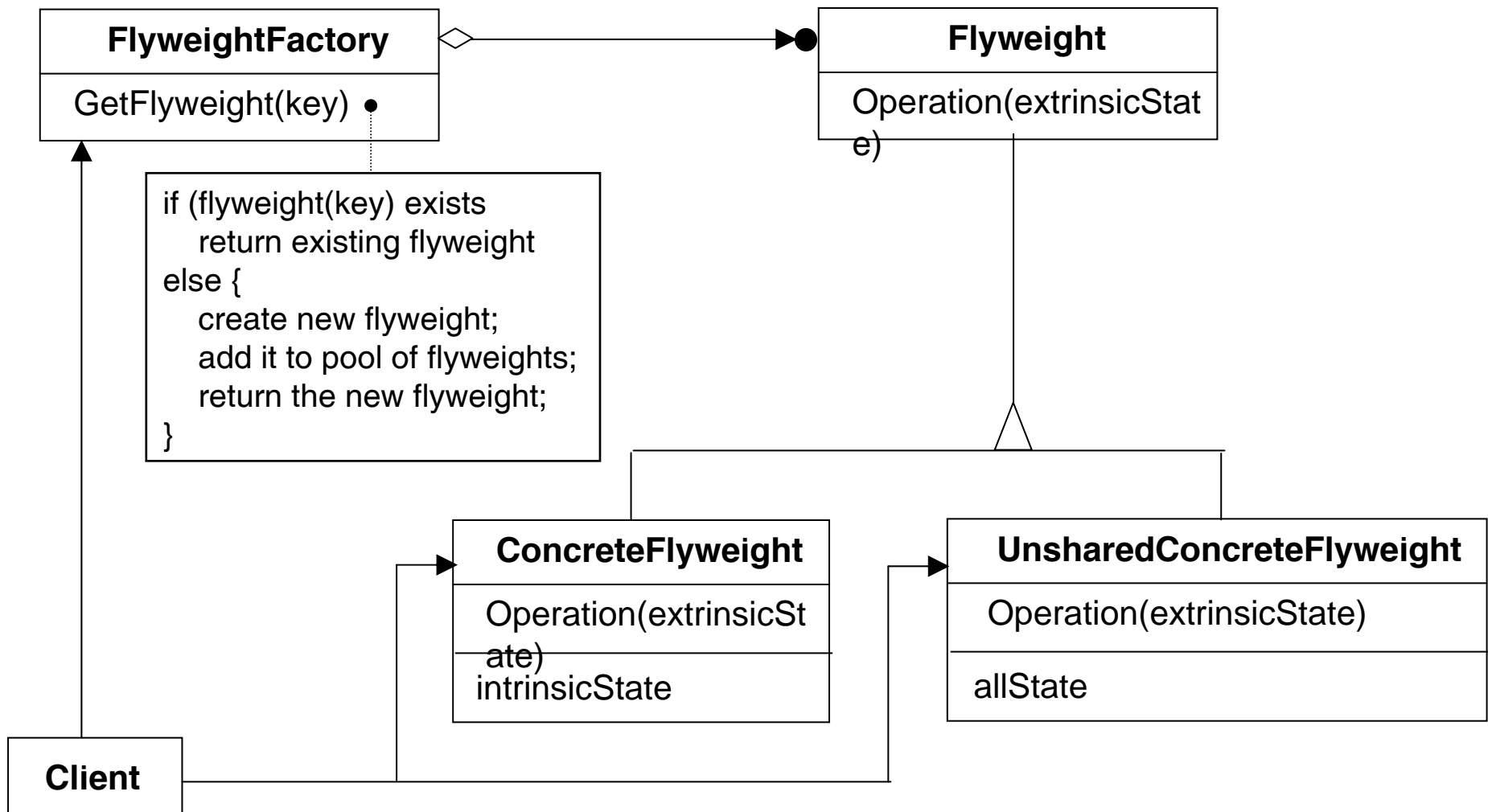
Mẫu Flyweight

- ❑ Mục tiêu : dùng phương tiện dùng chung để quản lý hiệu quả 1 số lớn đối tượng nhỏ.
- ❑ Nhu cầu áp dụng :
 - thiết kế hướng đối tượng thường có nhiều điểm lợi, nhưng hiện thực ấu trĩ bản thiết kế có thể trả giá đắt về sự không hiệu quả.
 - thí dụ chương trình xử lý văn bản có thể dùng khái niệm đối tượng để miêu tả bất kỳ phần tử cơ bản nào : ký tự, công thức, hình, Tuy nhiên ký tự là đối tượng rất nhỏ và xuất hiện rất nhiều lần trong văn bản, nếu mỗi lần xuất hiện 1 ký tự, ta tạo riêng 1 đối tượng mới cho ký tự đó thì rất không hiệu quả. Mẫu Flyweight rất thích hợp để giải quyết vấn đề dùng chung ký tự.
 - Flyweight là đối tượng dùng chung dựa vào khái niệm cơ bản là trạng thái trong và trạng thái ngoài. Trạng thái trong được chứa trong flyweight, độc lập với ngữ cảnh sử dụng (code ký tự,...). Trạng thái ngoài phụ thuộc và thay đổi theo ngữ cảnh, nó không thể được chứa trong flyweight, ngữ cảnh cần truyền cho flyweight trạng thái ngoài khi nhờ flyweight 1 công việc nào đó.

Thí dụ về mẫu Flyweight



Sơ đồ cấu trúc của mẫu Flyweight



Các phần tử tham gia

- ❑ Flyweight (Glyph)
 - định nghĩa interface cho đối tượng nhận yêu cầu và hoạt động theo trạng thái ngoài.
- ❑ ConcreteFlyweight (Character)
 - hiện thực interface Flyweight thành các đối tượng dùng chung.
- ❑ UnsharedConcreteFlyweight (Character)
 - hiện thực interface Flyweight thành các đối tượng không dùng chung.
- ❑ FlyweightFactory
 - tạo và quản lý các đối tượng Flyweight.
- ❑ Client
 - chứa tham khảo đến flyweight và nhờ khi cần

Các ngữ cảnh nên dùng mẫu Flyweight

Khi các điều kiện sau đồng thời thỏa mãn :

- ❑ ứng dụng dùng 1 số lớn đối tượng.
- ❑ giá bộ nhớ cao vì phải chứa số lượng lớn đối tượng.
- ❑ phần lớn trạng thái đối tượng có thể để bên ngoài.
- ❑ nhiều nhóm đối tượng có thể được thay thế bằng 1 số nhỏ đối tượng khi các trạng thái ngoài của chúng bị loại bỏ.
- ❑ ứng dụng không phụ thuộc vào tên nhận dạng đối tượng.



Chương 11

CÁC MẪU CREATIONAL

- Mẫu Abstract Factory
- Mẫu Factory Method
- Mẫu Prototype
- Mẫu Singleton
- Mẫu Builder



Creational Patterns

- ❑ Creational design patterns giúp xây dựng hệ thống linh động về mặt khởi tạo, quản lý và sử dụng đối tượng. Chúng có thể cho phép hệ thống chủ động trong việc xác định đối tượng nào được tạo, ai tạo ra đối tượng đó, cách thức và thời điểm khởi tạo đối tượng đó.
- ❑ Đặc điểm nổi bật trong creational patterns là chương trình cần sử dụng đối tượng không trực tiếp sinh ra đối tượng mà nhờ các phần tử trung gian để tăng độ linh động.
- ❑ Class creational patterns sử dụng đặc điểm thừa kế để thay đổi class sẽ được sử dụng để sinh ra đối tượng, Object creational patterns truyền quá trình khởi tạo đối tượng cho một đối tượng khác.

Ví dụ về quá trình khởi tạo đối tượng

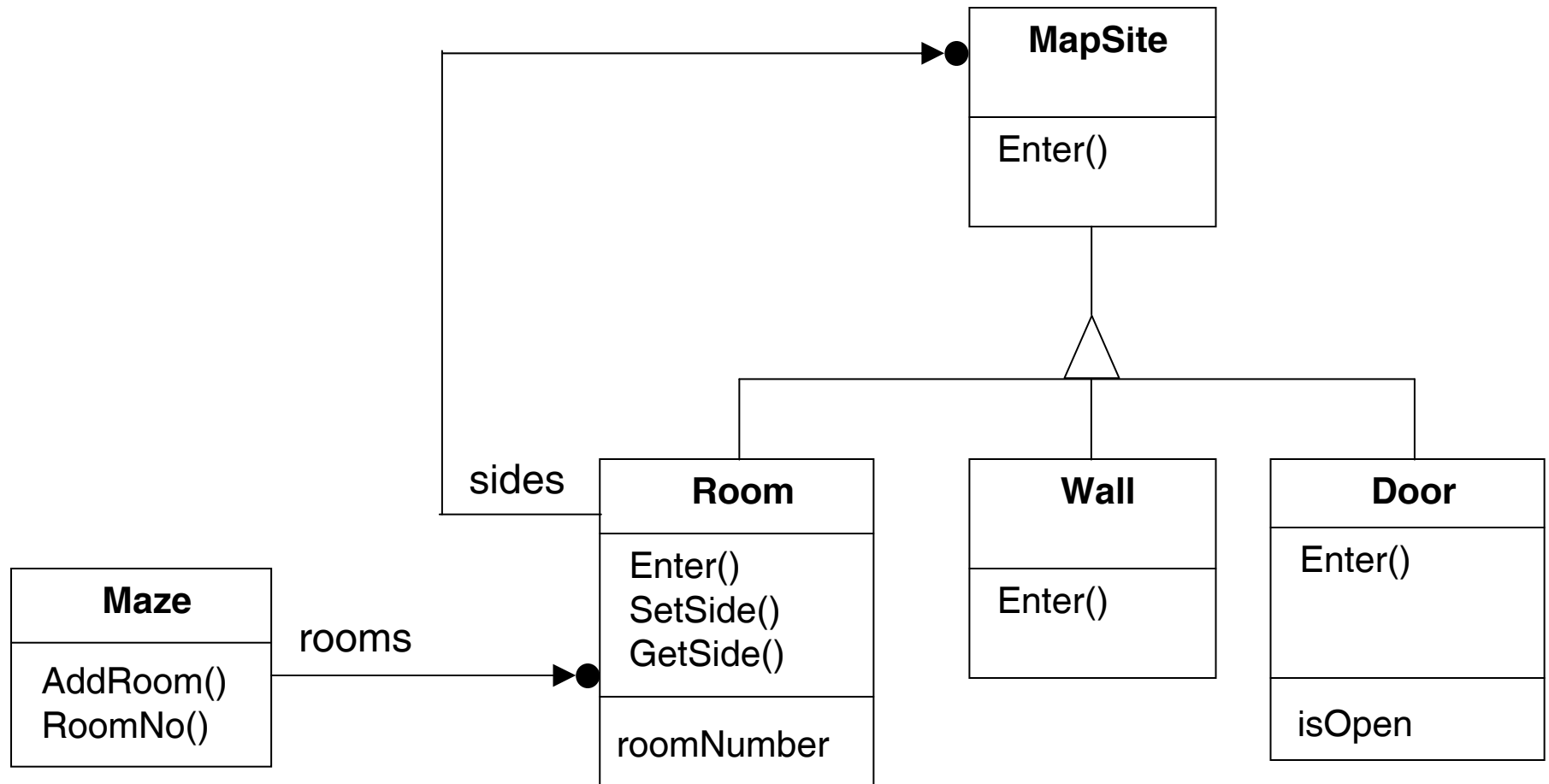
- ❑ Xây dựng một mê lộ (maze) cho các trò chơi có sử dụng mê lộ.
- ❑ Một mê lộ được định nghĩa bằng một tập hợp các phòng (room), mỗi phòng biết các đối tượng kế cận nó ở 4 hướng: bắc, nam, đông, tây. Đối tượng kế cận có thể là một phòng khác, một bức tường (wall) hay một cánh cửa (door) sang phòng khác.
- ❑ Các hướng có thể được hiện thực bởi các hằng số hay kiểu enum trong C++:

```
enum Direction {North , South , East , West }
```

- ❑ Lược đồ lớp của hệ thống như sau:



Ví dụ về quá trình khởi tạo đối tượng



Ví dụ về quá trình khởi tạo đối tượng

- ❑ MapSite là lớp cha trừu tượng của tất cả các phần tử trong mê lộ. MapSite chỉ có một phương thức Enter để chỉ thao tác đi vào một phần tử (room, door, wall). Tùy đặc điểm của mình, các phần tử sẽ phải override phương thức này.
- ❑ Lớp Room có thể được hiện thực trong C++ như sau :

```
class Room : public MapSite {  
    public:  
        Room (int roomNo);  
        MapSite*  GetSide (Direction) const;  
        void SetSide(Direction, MapSite*);  
        virtual void Enter();  
    private:  
        MapSite m_sides[4];  
        int m_roomNumber;  
};
```


Ví dụ về quá trình khởi tạo đối tượng

- ❑ Class Wall có thể được hiện thực trong C++ như sau :

```
class Wall : public MapSite {  
public:  
    Wall ();  
    virtual void Enter();  
};
```

- ❑ Class Door có thể hiện thực trong C++ như sau :

```
class Door : public MapSite {  
public:  
    Door (Room* = 0, Room*=0);  
    virtual void Enter();  
    Room* OtherSideFrom (Room*);  
private:  
    Room* m_room1, m_room2;  
    bool m_isOpen;  
};
```



Ví dụ về quá trình khởi tạo đối tượng

- ❑ Lớp Maze có thể hiện thực trong C++ như sau :

```
class Maze {  
public:  
    Maze();  
    void AddRoom(Room*);  
    Room* RoomNo(int) const;  
private:  
    //...  
};
```

- ❑ Lớp MazeGame tích hợp những lớp đã giới thiệu để tạo thành một game có sử dụng mê lộ. Lớp này phải tạo ra một mê lộ (Maze), method CreateMaze tạo một mê lộ đơn giản gồm 2 phòng có thể được định nghĩa trong C++ như sau :

Ví dụ về quá trình khởi tạo đối tượng

```
Maze* MazeGame::CreateMaze() {  
    Maze* aMaze=new Maze;  
    Room* r1=new Room(1);  
    Room* r2=new Room(2);  
    Door* theDoor = new Door(r1,r2);  
    aMaze->AddRoom(r1)  
    aMaze->AddRoom(r2)  
    r1->setSide(North, new Wall); ...  
    r2->setSide(North, new Wall); ...  
    return aMaze;  
}
```

- ❑ Phương thức CreateMaze bị ràng buộc cứng (Hard code), điều này dẫn đến hai nhược điểm :
 - Không thể sinh ra một maze có cấu trúc khác (VD: các phần tử ở các hướng của các room thay đổi).
 - Rất khó tái sử dụng phương thức này để tạo ra một maze có đặc điểm khác (VD: maze trong đó các phòng có thể có bom hoặc quà tặng, cửa giữa các phòng chỉ có thể mở bằng câu thần chú...) vì CreateMaze đã ràng buộc cứng các tên lớp.

Giải pháp khắc phục

- ❑ Truyền cho phương thức CreateMaze một đối tượng có khả năng sinh ra room, wall, door theo đặc thù của ứng dụng. Khi đó nếu muốn thay đổi maze thì chỉ cần truyền một đối tượng khác. Đây là hướng tiếp cận của mẫu **Abstract Factory**.
- ❑ Code trong CreateMaze gọi các phương thức thông thường (hoặc virtual) để khởi tạo đối tượng thay vì gọi constructor của lớp tương ứng. Khi đó nếu muốn thay đổi lớp sẽ tạo đối tượng thì chỉ cần tạo một subclass thừa kế MazeGame, trong đó override các phương thức khởi tạo đối tượng ở trên. Đây là hướng tiếp cận của mẫu **Factory Method**.
- ❑ CreateMaze được truyền các đối tượng room, door, wall có khả năng sinh ra đối tượng tương tự chúng (clone). Khi đó nếu muốn thay đổi đối tượng trong CreateMaze, ta chỉ cần truyền vào các đối tượng khác. Các đối tượng truyền cho CreateMaze gọi là Prototype và hướng tiếp cận này là của mẫu **Prototype**.
- ❑ CreateMaze được truyền 1 đối tượng mà có thể tạo mê lộ mới dùng các tác vụ thêm phòng, cửa và tường vào mê lộ mà nó xây dựng rồi ta dùng thừa kế để thay đổi các phần của mê lộ hay cách thức xây dựng mê lộ. Đây là hướng tiếp cận của mẫu **Builder**.

Trong các slide sau chúng ta sẽ tìm hiểu các mẫu phần mềm này.



Abstract Factory

- ❑ Mục tiêu : cung cấp interface cho việc khởi tạo đối tượng mà không cần xác định trước lớp cụ thể (concrete) tương ứng.
- ❑ Nhu cầu áp dụng : có những trường hợp khi xây dựng chương trình chúng ta chưa biết chính xác hay chưa muốn ràng buộc lớp nào sẽ được sử dụng để sinh ra đối tượng, chẳng hạn:
 - chương trình có khả năng chạy trên nhiều platform. Mỗi platform có một họ các lớp giao diện, việc sử dụng cụ thể họ lớp giao diện nào chỉ biết khi chương trình chạy.
 - framework cần khởi tạo đối tượng nhưng chưa biết trước lớp cụ thể sẽ sử dụng.

Những chương trình như vậy thường có một số yêu cầu đối với người thiết kế:

- code chương trình phải có khả năng tương tác tổng quát lên các đối tượng sẽ được sinh ra.
- dễ chuyển đổi giữa các họ đối tượng.
- dễ bổ sung các họ đối tượng mới.

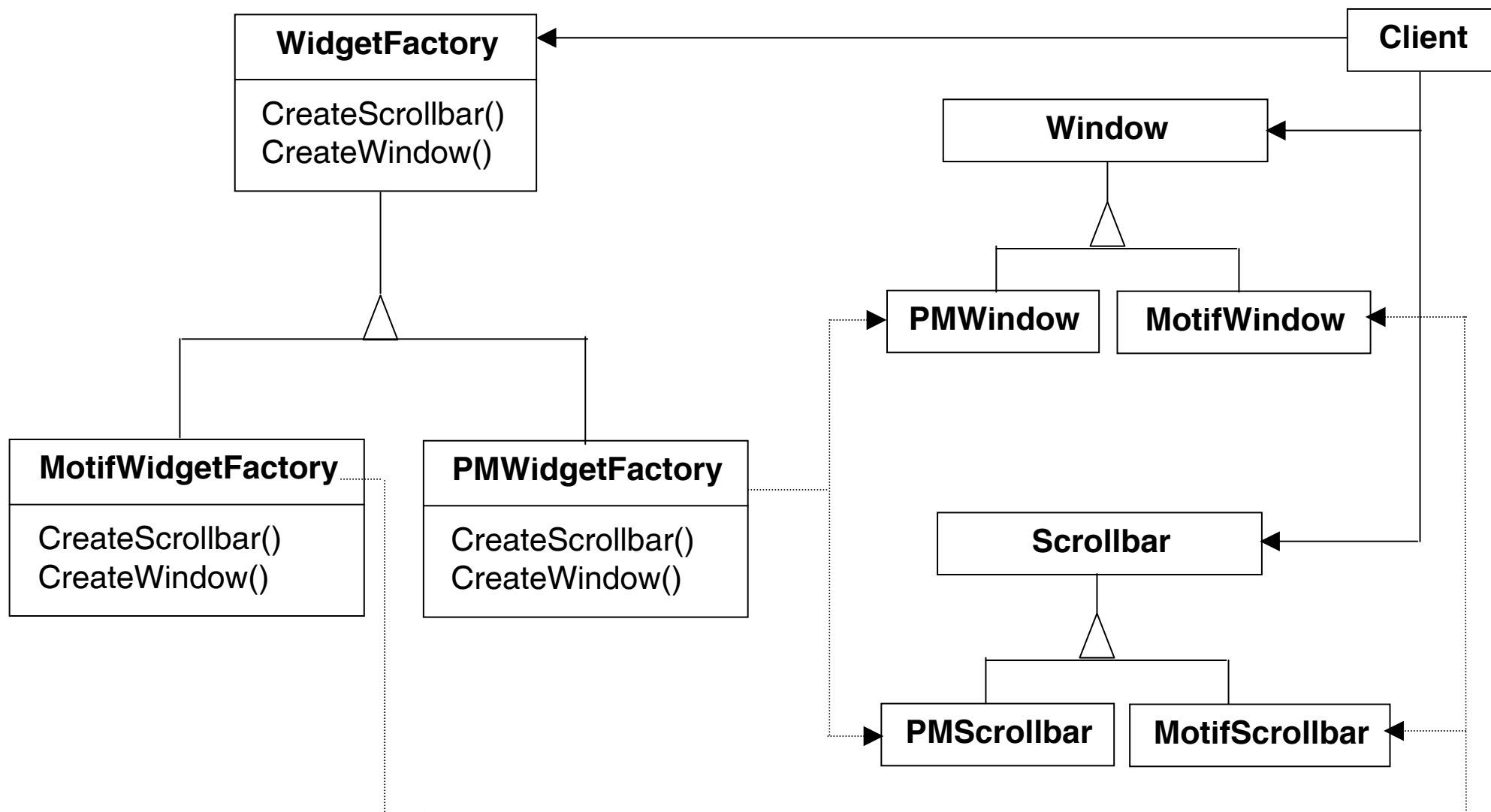
Ví dụ về quá trình khởi tạo đối tượng

- ❑ Giải pháp đề nghị:
 - Cung cấp interface cho các đối tượng dự định sẽ được khởi tạo khi hệ thống chạy, gọi là `AbstractProduct`.
 - Cung cấp interface để khởi tạo các đối tượng kiểu `AbstractProduct`, gọi là `AbstractFactory`.
 - Để khởi tạo một đối tượng cụ thể, cần xây dựng 2 lớp concrete: một lớp hiện thực interface `AbstractFactory` để khởi tạo đối tượng từ lớp hiện thực `AbstractProduct`.

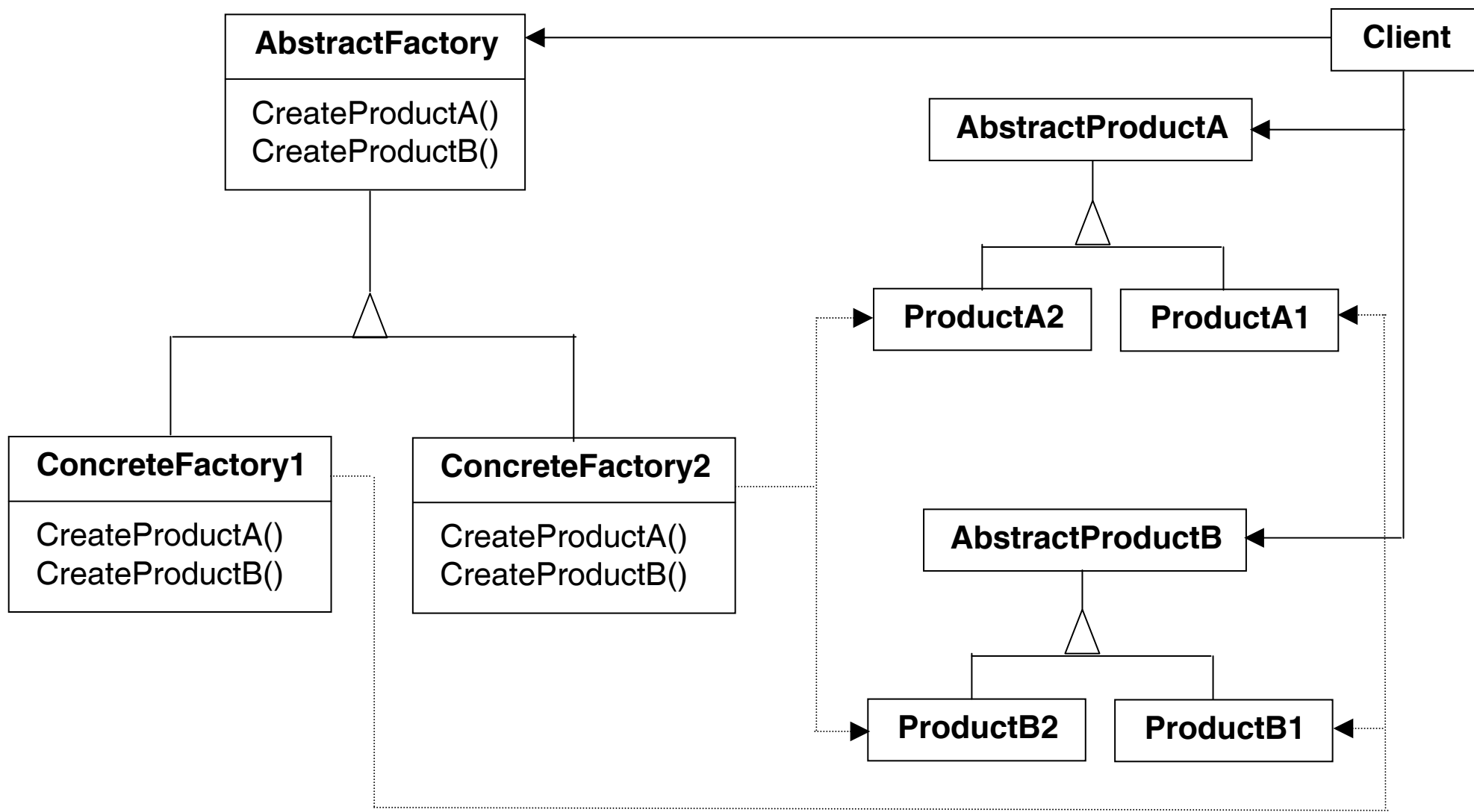
Hướng tiếp cận trên là của mẫu `Abstract Factory`.



Thí dụ về mẫu AbstractFactory



Sơ đồ cấu trúc của mẫu AbstractFactory



Các phần tử tham gia

- ❑ **AbstractFactory** : định nghĩa interface cho việc khởi tạo đối tượng. Thường là lớp Abstract.
- ❑ **ConcreteFactory** : hiện thực các method trong interface AbstractFactory để tạo ra các đối tượng cụ thể. Hệ thống có nhiều ConcreteFactory, mỗi ConcreteFactory sinh ra một nhóm đối tượng, các nhóm đối tượng do các ConcreteFactory sinh ra tương đồng nhau về vai trò.
- ❑ **AbstractProduct** : lớp Abstract, định nghĩa interface cho một kiểu lớp (kiểu lớp button, kiểu lớp Scrollbar...)
- ❑ **ConcreteProduct** : lớp hiện thực đối tượng được sinh ra từ lớp ConcreteFactory tương ứng, thừa kế lớp Abstract Product.

Các phần tử tham gia

- ❑ **Client** : chương trình cần tạo các đối tượng. Client chỉ sử dụng các interface `AbstractFactory` và `AbstractProduct`.
- ❑ Quá trình tương tác giữa các phần tử :
 - Tại thời điểm compile, Client nắm giữ pointer đến phần tử của lớp `AbstractFactory` (giả sử là `_factory`).
 - Tại thời điểm run-time, Client biết cần sử dụng họ đối tượng `ConcreteProduct` nào sẽ khởi tạo đối tượng `ConcreteFactory` tương ứng và gán vào `_factory`. Client thông qua interface `AbstractFactory` để yêu cầu đối tượng `ConcreteFactory` sinh ra các đối tượng mong muốn.
 - Client dựa vào interface của các đối tượng `ConcreteProduct` được khai báo trong `AbstractProduct` để sử dụng các đối tượng này.

Các ngữ cảnh nên dùng mẫu Abstract Factory

- ❑ hệ thống muốn xác định quá trình khởi tạo và sử dụng đối tượng tại thời điểm chạy chương trình.
- ❑ hệ thống muốn tương tác với một họ trong một tập hợp họ đối tượng và việc chọn họ đối tượng được xác định tại thời điểm run-time.
- ❑ hệ thống muốn ràng buộc tính sử dụng đồng thời các phần tử trong một họ đối tượng.



Ví dụ áp dụng

- ❑ Lớp MazeFactory (tương ứng phần tử Abstract Factory) cung cấp phương thức mặc định để khởi tạo các đối tượng maze, wall, room, door. Các lớp con thừa kế có thể override các phương thức này để sinh ra các maze, wall, room, door khác.
- ❑ MazeFactory không sử dụng Constructor mà cung cấp các method Make để tạo ra các đối tượng.
- ❑ MazeFactory có thể được hiện thực trong C++ như sau:

```
class MazeFactory {  
    public:        MazeFactory();  
        virtual Maze* MakeMaze() const { return new Maze;}  
        virtual Wall* MakeWall() const { return new Wall;}  
        virtual Room* MakeRoom(int n) const {  
            return new Room(n); }  
        virtual Door* MakeDoor(Room* r1, Room* r2) const {  
            return new Door(r1, r2);  
        }  
}
```

Ví dụ áp dụng

- ❑ Lớp MazeGame đóng vai trò là Client sử dụng mẫu Abstract Factory. Phương thức CreateMaze được truyền một đối tượng kiểu MazeFactory, CreateMaze sử dụng đối tượng này để sinh ra các đối tượng cần thiết.
- ❑ CreateMaze có thể được hiện thực trong C++ như sau:

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {  
    Maze* aMaze = factory.MakeMaze();  
    Room* r1 = factory.MakeRoom (1);  
    Room* r2 = factory.MakeRoom (2);  
    Door* aDoor = factory.MakeDoor(r1, r2);  
    aMaze->AddRoom(r1);  
    aMaze->AddRoom(r2);  
    r1->SetSide(North, factory.MakeWall());  
    ...  
    return aMaze;  
}
```

Ví dụ áp dụng

- ❑ Để thay đổi đối tượng tạo ra, ta chỉ việc truyền một đối tượng factory khác, miễn là đối tượng này thuộc kiểu MazeFactory.
- ❑ VD để tạo một maze mà trong room có thể có bom, ta định nghĩa lớp RoomWithABomb thừa kế lớp Room và lớp BombedMazeFactory thừa kế lớp MazeFactory trong đó override phương thức MakeRoom như sau:

```
Room* BombedMazeFactory::MakeRoom(int n) const {  
    return new RoomWithABomb(n);  
}
```

Trong chương trình Game chỉ cần thực hiện đoạn code:

```
MazeGame game;  
BombedMazeFactory factory;  
game.CreateMaze(factory);
```

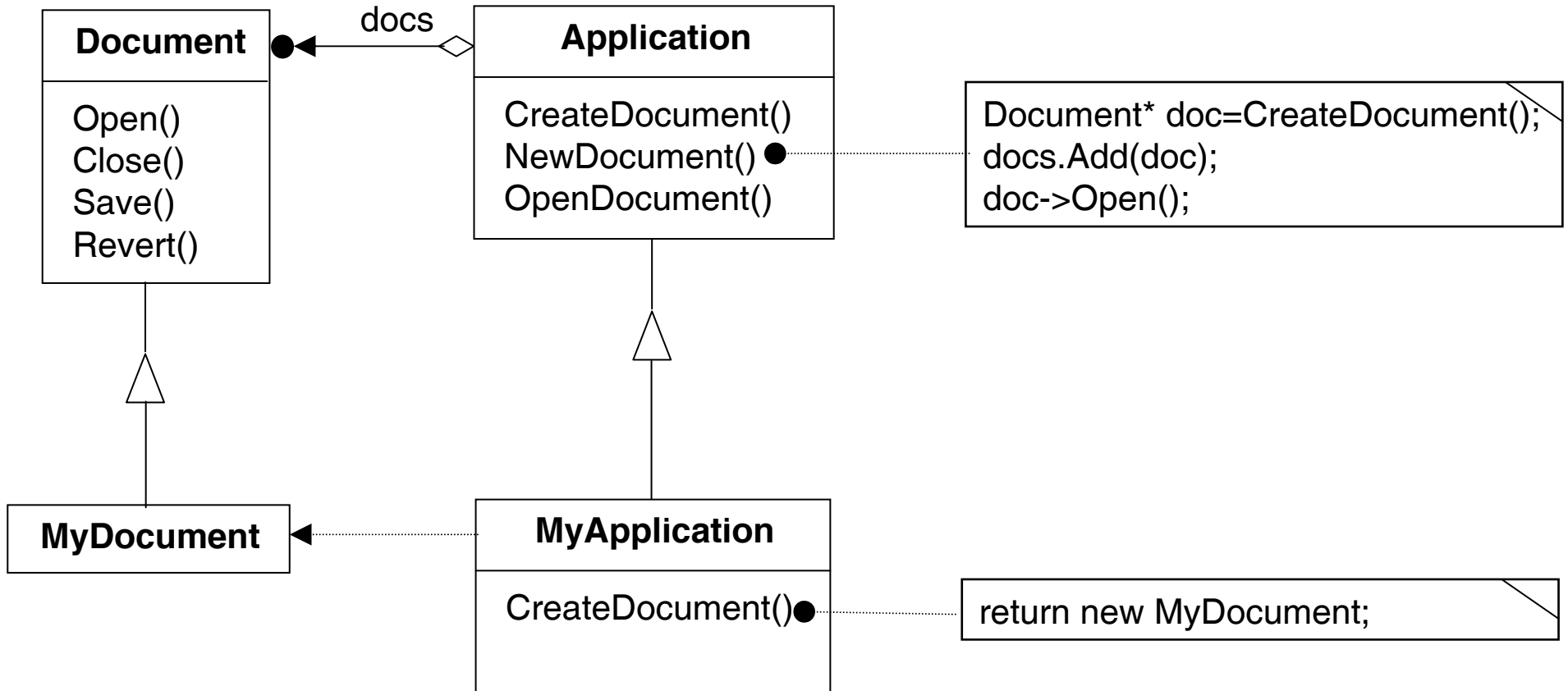
Ví dụ áp dụng

- ❑ Khi áp dụng mẫu Abstract Factory, chương trình có một số đặc điểm :
 - có thể thích ứng với nhiều kiểu maze khác nhau.
 - việc thêm một kiểu maze chỉ là thêm một họ các lớp, không ảnh hưởng đến code chương trình đã có.
 - chương trình nhất quán trong việc sử dụng một họ các phần tử của cùng một kiểu maze, không xảy ra trường hợp 2 phần tử của 2 họ khác nhau cùng tồn tại trong code của client.
 - việc bổ sung một loại phần tử vào maze sẽ gặp khó khăn vì phải bổ sung vào lớp MazeFactory, do đó sẽ ảnh hưởng đến hầu như tất cả các lớp khác.

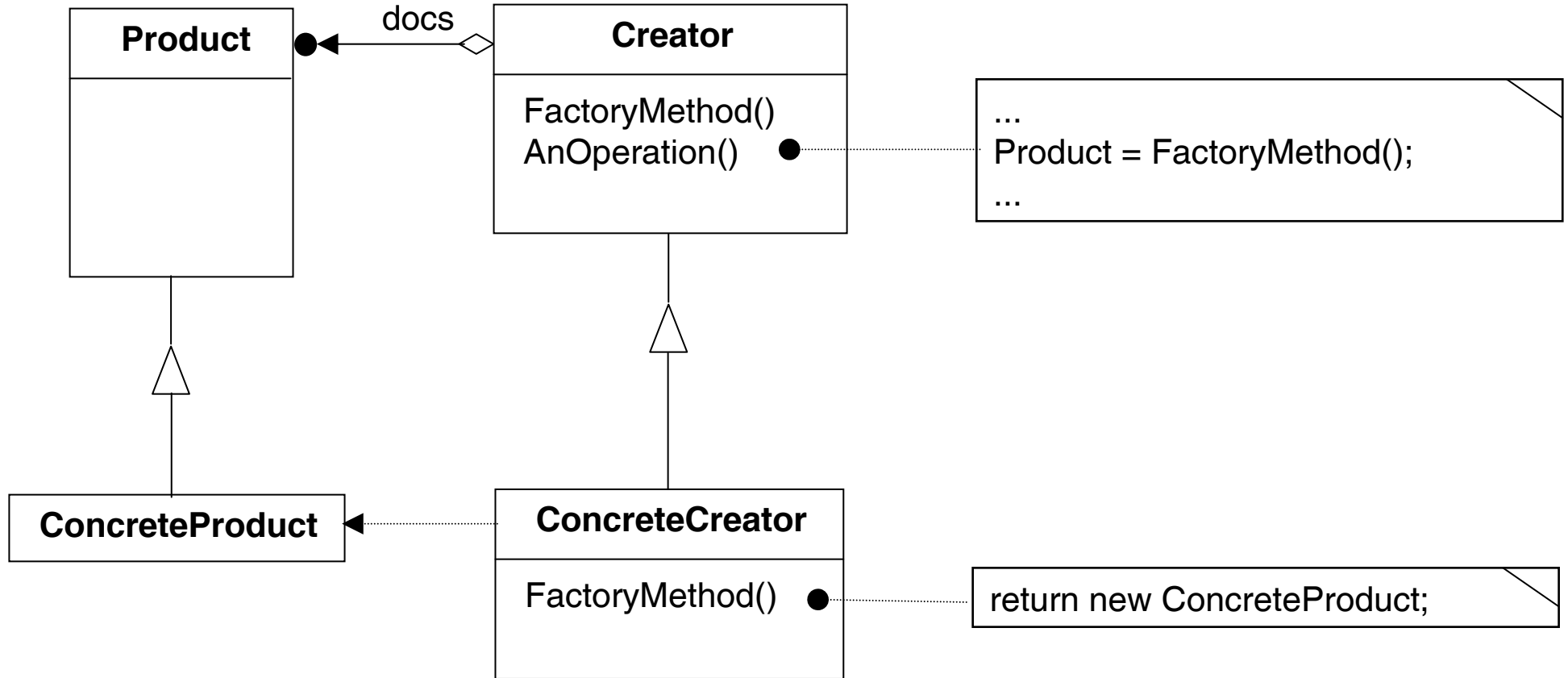
Factory Method

- ❑ Mục tiêu : định nghĩa interface để sinh ra đối tượng nhưng để cho lớp con quyết định lớp nào được dùng để sinh ra đối tượng. Factory Method cho phép một lớp chuyển quá trình khởi tạo đối tượng cho lớp con.
- ❑ Nhu cầu áp dụng :
 - Tương tự nhu cầu áp dụng ở mẫu Abstract Factory: cần phải cung cấp giải pháp tạo đối tượng nhưng chưa biết được lớp cụ thể dùng để sinh ra đối tượng.
 - Hướng giải quyết theo mẫu Factory Method: thay vì tạo lớp Abstract Factory để tạo các đối tượng như trong mẫu Abstract Factory, Factory Method tạo một phương thức ảo, các lớp con sẽ override phương thức này để khởi tạo đối tượng.

Sơ đồ thí dụ



Sơ đồ cấu trúc của mẫu Factory method



Các phần tử tham gia

- ❑ **Product** : định nghĩa interface cho các đối tượng sản phẩm.
- ❑ **ConcreteProduct** : lớp thể hiện đối tượng sản phẩm cần tạo. Hiện thực interface Product.
- ❑ **Creator** : định nghĩa factory method, sản phẩm trả về là đối tượng kiểu Product.
- ❑ **ConcreteCreator** : thừa kế lớp Creator, override factory method để trả về đối tượng ConcreteProduct.



Các ngữ cảnh nên dùng mẫu Factory Method

- ❑ một lớp không biết trước lớp của đối tượng mà nó cần khởi tạo.
- ❑ một lớp muốn lớp con của mình thay đổi hay xác định lớp của đối tượng cần khởi tạo.
- ❑ một lớp muốn chuyển quá trình hiện thực một nhiệm vụ nào đó cho một trong các lớp con nhưng cho phép ứng dụng xác định lớp con cụ thể.



Ví dụ áp dụng

- ❑ Lớp MazeGame đóng vai trò là Creator, cung cấp các factory method để tạo các đối tượng:

```
class MazeGame {  
    public: Maze* CreateMaze();  
    // factory methods:  
    virtual Maze* MakeMaze() const { return new Maze; }  
    virtual Room* MakeRoom(int n) const { return new Room(n); }  
    virtual Wall* MakeWall() const { return new Wall; }  
    virtual Door* MakeDoor(Room* r1, Room* r2) const { return  
        new Door(r1, r2); }  
};
```



Ví dụ áp dụng

- ❑ Method CreateMaze có thể được định nghĩa lại như sau:

```
Maze* MazeGame::CreateMaze () {  
    Maze* aMaze = MakeMaze();  
    Room* r1 = MakeRoom(1);    Room* r2 = MakeRoom(2);  
    Door* theDoor = MakeDoor(r1, r2);  
    aMaze->AddRoom(r1);  
    aMaze->AddRoom(r2);  
    r1->SetSide(North, MakeWall());  
    //.....  
    r2->SetSide(North, MakeWall());  
    //.....  
    return aMaze;  
}
```



Ví dụ áp dụng

❑ Các maze game khác muốn thay đổi phần tử trong game thì:

- Định nghĩa các lớp thừa kế từ các lớp thể hiện các phần tử tương ứng

VD: RoomWithABomb thừa kế Room.

- Định nghĩa lớp thừa kế lớp MazeGame, trong lớp này override factory method tạo ra đối tượng muốn thay đổi.

VD:

```
class BombedMazeGame : public MazeGame {  
    public: BombedMazeGame();  
    virtual Wall* MakeWall() const { return new BombedWall; }  
    virtual Room* MakeRoom(int n) const  
        { return new RoomWithABomb(n); }  
};
```



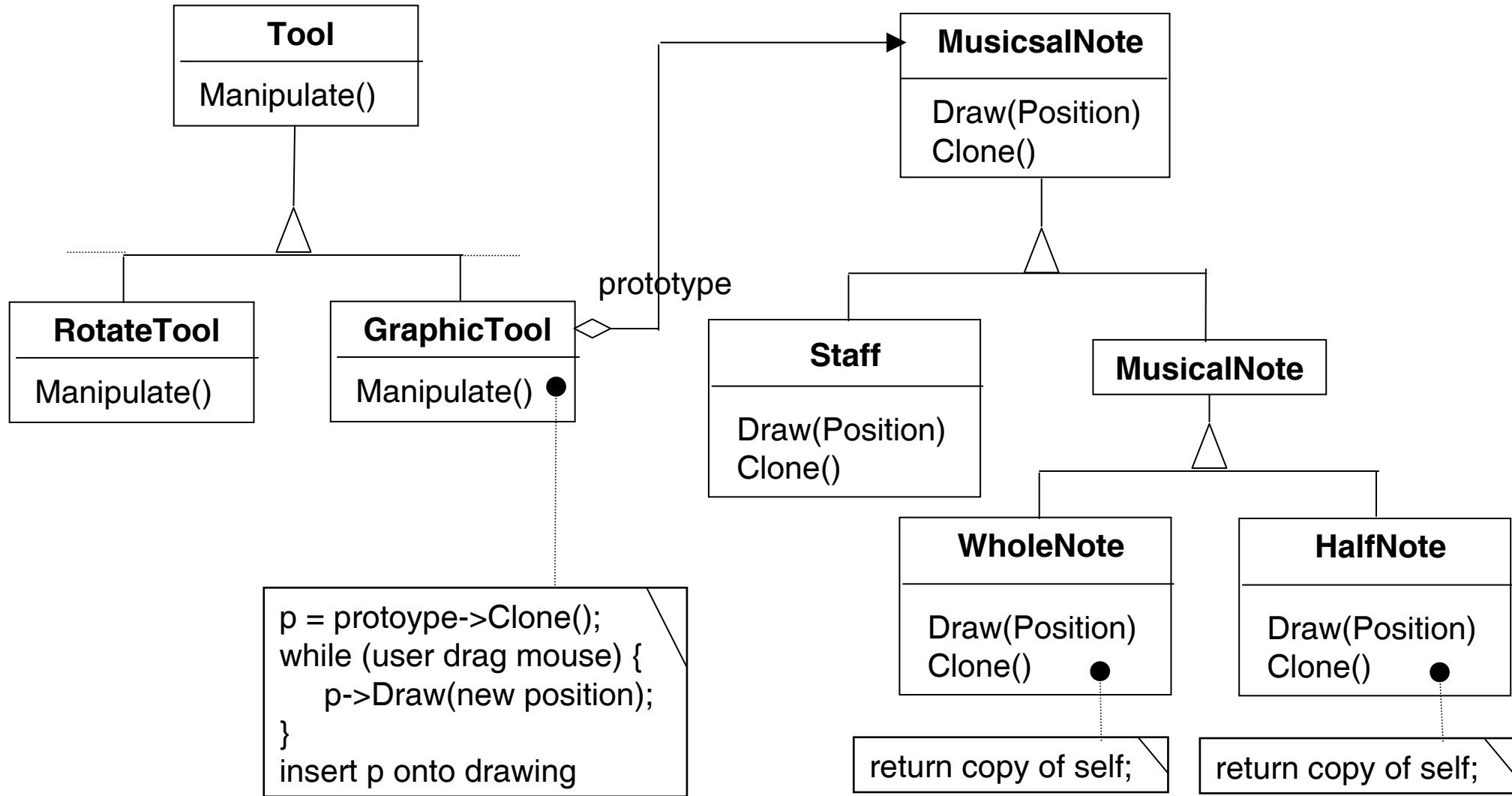
Prototype

- ❑ Mục tiêu : giúp khởi tạo đối tượng bằng cách copy một đối tượng khác (prototype) đang tồn tại.
- ❑ Nhu cầu áp dụng :
 - Muốn tạo đối tượng nhưng không biết hoặc không muốn sử dụng lớp Concrete. Ví dụ Editor Framework cho phép ứng dụng bổ sung các control vào toolbox nhưng chưa biết lớp cụ thể sẽ sinh ra Control.
 - Giải pháp đề nghị :
 - + Framework cung cấp interface để copy một đối tượng, interface này sẽ được các lớp concrete cần sinh ra đối tượng implement.
 - + Ứng dụng phát triển từ Framework hay component tạo ra đối tượng mong muốn, sau đó mỗi lần tạo thêm một đối tượng bằng cách gọi hàm copy đối tượng này.

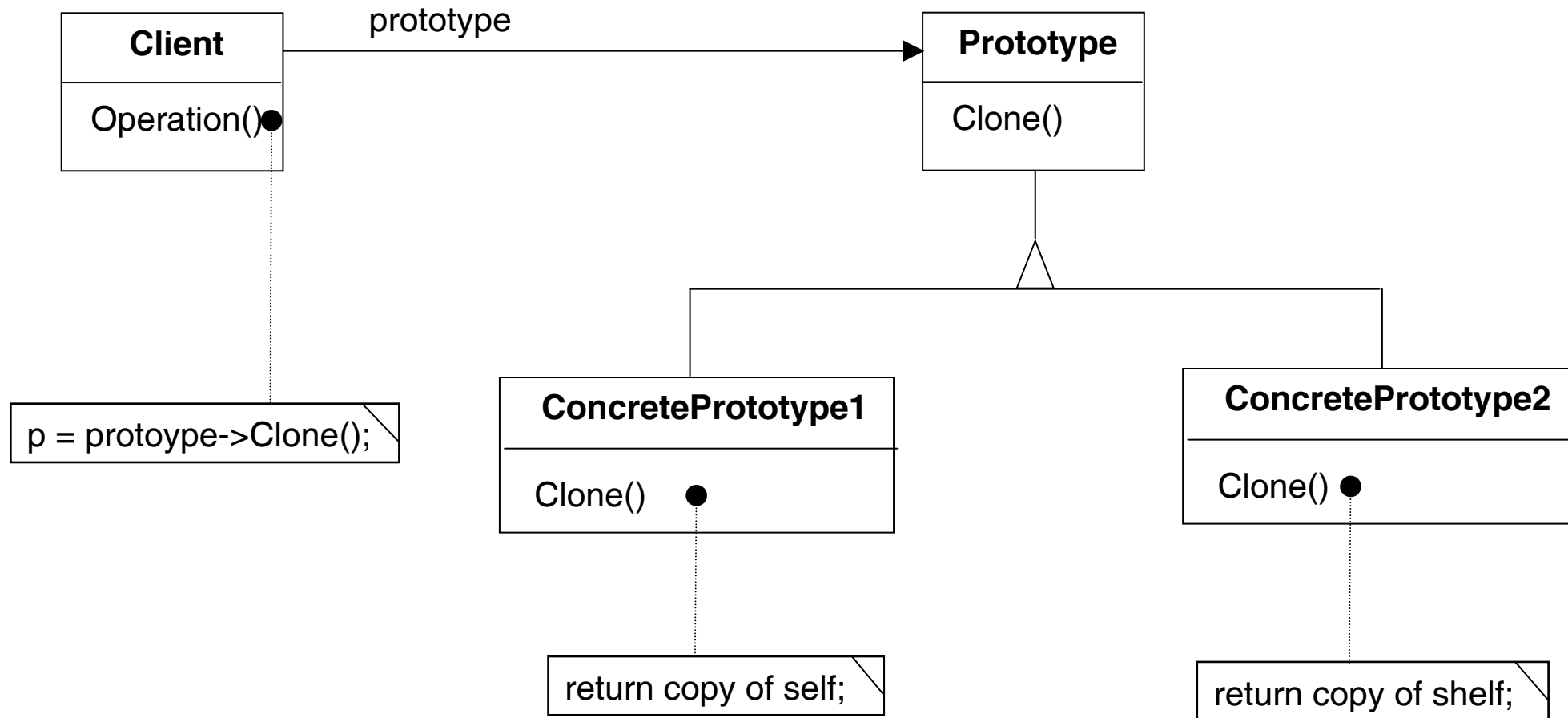
Đây chính là phương pháp của mẫu Prototype.



Ví dụ về mẫu Prototype



Sơ đồ cấu trúc của mẫu Prototype



Các phần tử tham gia

- ❑ Prototype: cung cấp interface để copy chính nó (clone)
- ❑ ConcretePrototype: hiện thực interface được cung cấp bởi Prototype để copy chính nó.
- ❑ Client: tạo mới đối tượng bằng cách yêu cầu một đối tượng đã có (prototype) copy chính nó.



Một số vấn đề hiện thực

- ❑ Class có thể cung cấp phương thức để set giá trị cho đối tượng sau khi được tạo ra bằng cách copy từ prototype.
- ❑ Nếu số lượng prototype trong ứng dụng không cố định, nên quản lý chúng bằng một đối tượng prototype manager. Đối tượng này chứa các tham khảo đến các prototype và các key tương ứng để truy xuất chúng.
- ❑ Việc hiện thực phương thức clone phụ thuộc nhiều vào ngôn ngữ lập trình. Có thể copy hoàn toàn (deep copy) hay share biến (shallow copy).
- ❑ Java, SmallTalk, Eiffel cung cấp phương thức clone(). C++ có phương thức copy.
- ❑ Tất cả những phương thức trên đều mặc định thực hiện shallow copy.
- ❑ Khi hiện thực deep copy, để ý vấn đề tham khảo vòng.

Các ngữ cảnh nên dùng mẫu Prototype

- ❑ Mẫu Prototype thường được sử dụng khi hệ thống cần độc lập với các đối tượng mà nó sinh ra và
 - khi lớp cần dùng để sinh đối tượng được xác định tại thời điểm chương trình chạy (dynamic loading), hoặc
 - tránh trường hợp xây dựng số lượng các phần tử khởi tạo đối tượng ngang bằng với số lượng kiểu đối tượng bổ sung dự định tạo ra, hoặc
 - khi các đối tượng của cùng một class có ít điểm khác biệt.



Ví dụ áp dụng

- ❑ Chương trình này phát triển từ ví dụ áp dụng mẫu Abstract Factory.
- ❑ Định nghĩa lớp MazePrototypeFactory thừa kế lớp MazeFactory, đối tượng lớp này được cung cấp các prototype để tạo ra các đối tượng cùng loại. MazePrototypeFactory có thể được hiện thực trong C++ như sau:

```
class MazePrototypeFactory : public MazeFactory{
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);
    virtual Maze* MakeMaze() const;
    ...// Các method khởi tạo các phần tử của maze
private:
    Maze*          _mazePrototype;
    Room*          _roomPrototype;
    Wall*          _wallPrototype;
    Door*          _doorPrototype;
}
```

Ví dụ áp dụng

- ❑ Constructor của lớp MazePrototypeFactory chỉ khởi tạo các prototype với các đối tượng tương ứng được cung cấp từ ứng dụng:

```
MazePrototypeFactory::MazePrototypeFactory (  
    Maze* m, Wall* w, Room* r, Door* d ) {  
    _mazePrototype = m;  
    _wallPrototype = w;  
    _roomPrototype = r;  
    _doorPrototype = d;  
}
```

Ví dụ áp dụng

- ❑ Các lớp Door, Wall, Room đều phải hiện thực phương thức clone thừa kế từ lớp MapSite và các phương thức set giá trị nếu cần:

```
class Door : public MapSite {  
    public: Door();  
           Door(const Door&);  
           virtual void Initialize(Room*, Room*);  
           virtual Door* Clone() const;  
    private: Room* _room1; Room* _room2;  
};  
Door::Door (const Door& other)  
    { _room1 = other._room1; _room2 = other._room2; }  
void Door::Initialize (Room* r1, Room* r2)  
    { _room1 = r1; _room2 = r2; }  
Door* Door::Clone () const { return new Door(*this); }
```


Ví dụ áp dụng

- ❑ Các phương thức khởi tạo wall, room, door đều clone prototype có sẵn và set các giá trị nếu cần:

```
Wall* MazePrototypeFactory::MakeWall () const {  
    return _wallPrototype->Clone();  
}
```

```
Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {  
    Door* door = _doorPrototype->Clone();  
    door->Initialize(r1, r2);  
    return door;  
}
```

Ví dụ áp dụng

- ❑ Để thay đổi maze cần tạo, ta chỉ cần cung cấp các factory cho constructor của MazePrototypeFactory :

MazeGame game;

MazePrototypeFactory bombedMazeFactory(new Maze, new BombedWall, new RoomWithABomb, new Door);

Maze* maze = game.CreateMaze(bombedMazeFactory);

- ❑ Khi bổ sung phần tử sản phẩm mới (VD: roomWithABomb), không cần phải cung cấp lớp Factory (BombedMazeFactory) để sinh ra các sản phẩm này mà chỉ cần cung cấp prototype của sản phẩm đó.

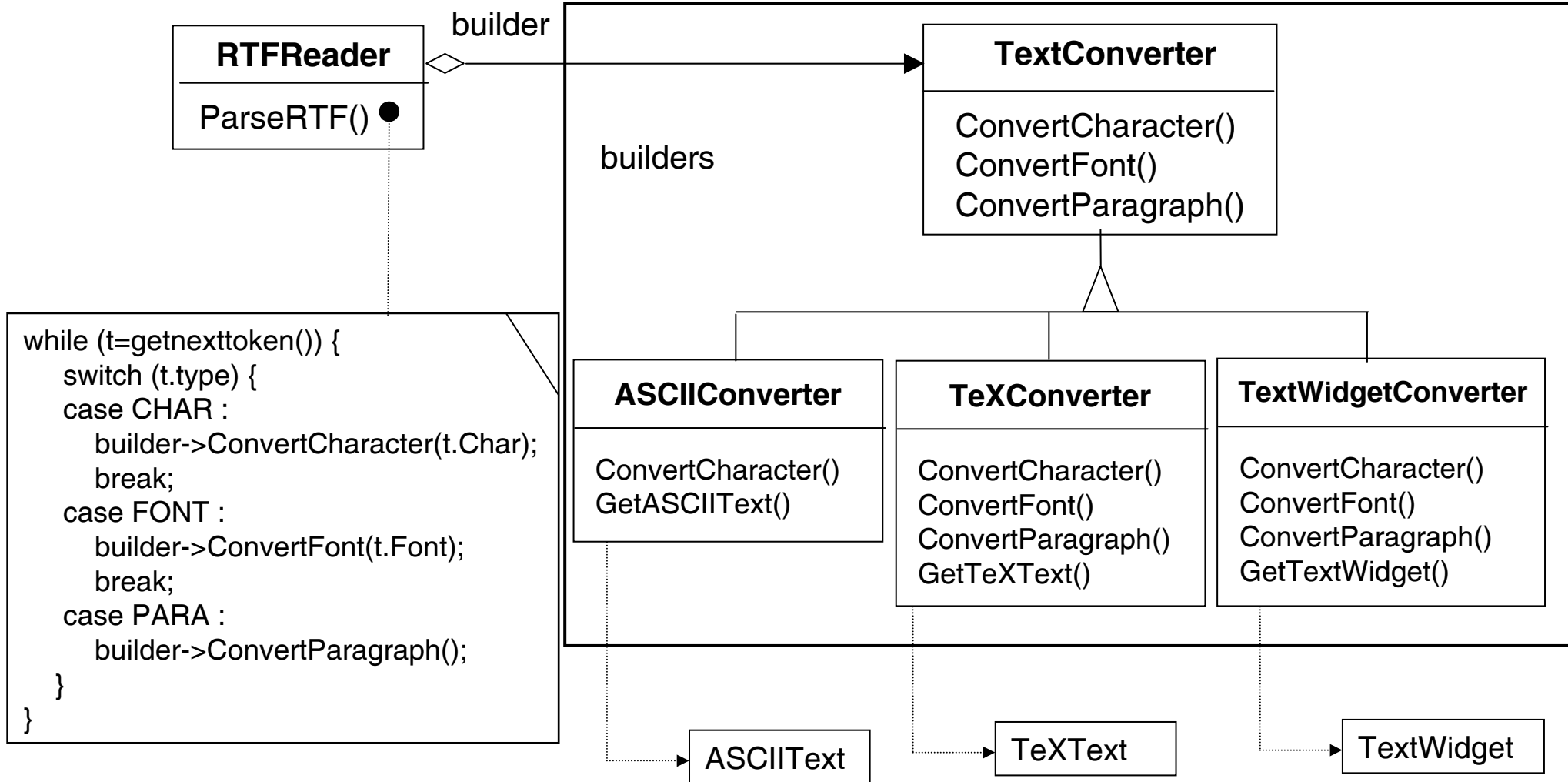
Mẫu Builder

- ❑ Mục tiêu : tách việc xây dựng đối tượng phức tạp khỏi việc miêu tả nó sao cho cùng 1 qui trình xây dựng có thể tạo ra nhiều sự miêu tả khác nhau.
- ❑ Nhu cầu áp dụng :
 - Trình đọc file RTF cần đổi file RTF sang 1 số dạng khác chưa biết trước như text thô, Tex,... hay sang 1 điều khiển soạn thảo text.
 - Giải pháp đề nghị :
 - + định nghĩa class TextConverter chứa các tác vụ chuyển đổi token cơ bản.
 - + Mỗi định dạng cần chuyển tới sẽ được đảm trách bởi 1 subclass của class TextConverter.

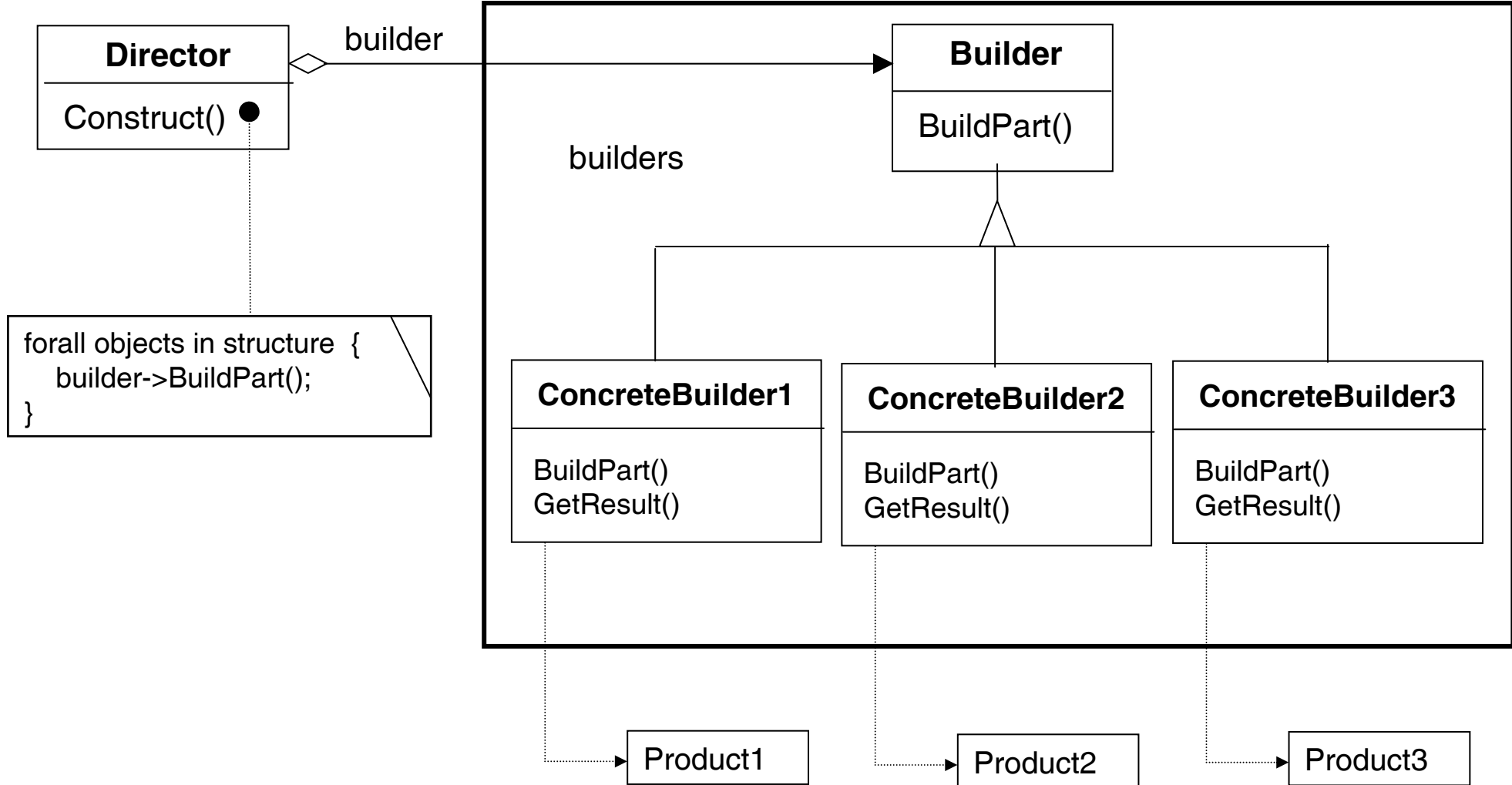
Đây chính là phương pháp của mẫu Builder.



Ví dụ về mẫu Builder



Sơ đồ cấu trúc của mẫu Builder



Các phần tử tham gia

- ❑ **Director** (RTFReader) : xây dựng đối tượng dùng interface Builder.
- ❑ **Builder** (TextConverter) : cung cấp interface để xây dựng các phần của đối tượng Product.
- ❑ **ConcreteBuilder** (ASCIIConverter, TeXConverter) :
 - xây dựng và lắp ghép các phần của Product bằng cách hiện thực interface của class Builder.
 - định nghĩa và ghi giữ đối tượng mà nó tạo ra.
 - cung cấp interface để nhận đối tượng.
- ❑ **Product** (ASCIIText, TeXText) :
 - miêu tả đối tượng phức tạp cần xây dựng.
 - bao gồm các class định nghĩa các thành phần bao gồm interface phục vụ việc lắp ghép các thành phần thành kết quả cuối cùng.

Các ngữ cảnh nên dùng mẫu Builder

- ❑ Mẫu Builder thường được sử dụng khi :
 - giải thuật tạo đối tượng phức tạp nên độc lập với các phần cấu thành đối tượng và cách lắp ghép chúng.
 - qui trình xây dựng phải cho phép xây dựng nhiều biến thể khác nhau của đối tượng



Ví dụ áp dụng

- ❑ Ta xây dựng interface của builder bằng class MazeBuilder sau :

```
class MazeBuilder {  
    protected:  
        MazeBuilder();  
    public:  
        // methods:  
        virtual void BuildMaze() { };  
        virtual void BuildRoom (int n) { };  
        virtual void BuildDoor (int roomFrom, int RoomTo) { } ;  
        virtual Maze* GetMaze() { return 0; };  
};
```



Ví dụ áp dụng

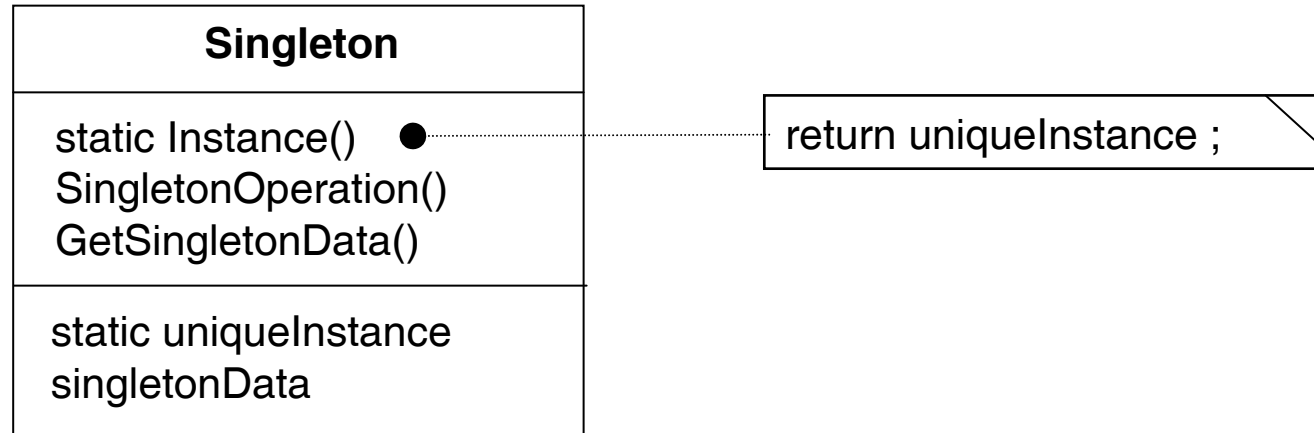
- ❑ Ta hiệu chỉnh hàm CreateMaze() của class MazeGame thành :

```
Maze* Mazegame::CreateMaze (MazeBuilder& builder) {  
    builder.BuildMaze();  
    builder.BuildRoom(1);  
    builder.BuildRoom(2);  
    builder.BuildDoor(1,2);  
    return builder.GetMaze();  
}
```
- ❑ Ta có thể định nghĩa các subclass của class MazeBuilder để tạo ra các Maze khác nhau và tạo đối tượng của subclass này rồi truyền nó như là tham số của hàm CreateMaze() để tạo các Maze khác nhau.

Singleton

- ❑ Mục tiêu : đảm bảo mỗi class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến đối tượng.
- ❑ Nhu cầu áp dụng :
 - chỉ 1 "printer spooler" để quản lý các máy in.
 - chỉ 1 trình quản lý cho các file của hệ thống file.
 - chỉ 1 trình quản lý windows cho các cửa sổ trên màn desktop...

Sơ đồ cấu trúc của mẫu Singleton



Các phần tử tham gia

❑ Singleton :

- định nghĩa tác vụ Instance giúp client truy xuất instance duy nhất của class. Instance() là tác vụ chung của class (hàm static trong C++).
- chịu trách nhiệm về việc tạo instance duy nhất cho class.

❑ **Cộng tác giữa các đối tượng** : các clients truy xuất instance của class Singleton thông qua việc gọi tác vụ Instance() của class.



Chương 12

CÁC MẪU BEHAVIORAL

- Mẫu Chain of Responsibility
- Mẫu Template Method
- Mẫu Strategy
- Mẫu Command
- Mẫu State
- Mẫu Observer

Behavioral Patterns

- ❑ Tập trung vào giải thuật và sự phân bố công việc giữa các object
- ❑ Class patterns sử dụng thừa kế để chuyển giao thao tác giữa các class.
- ❑ Object patterns sử dụng tính đa hình và bao gộp để truyền thao tác từ đối tượng này sang đối tượng khác.
- ❑ Các slide sau sẽ giới thiệu các pattern Chain of Responsibility, Template Method (class pattern), Strategy (object pattern) và Command (object Pattern),...

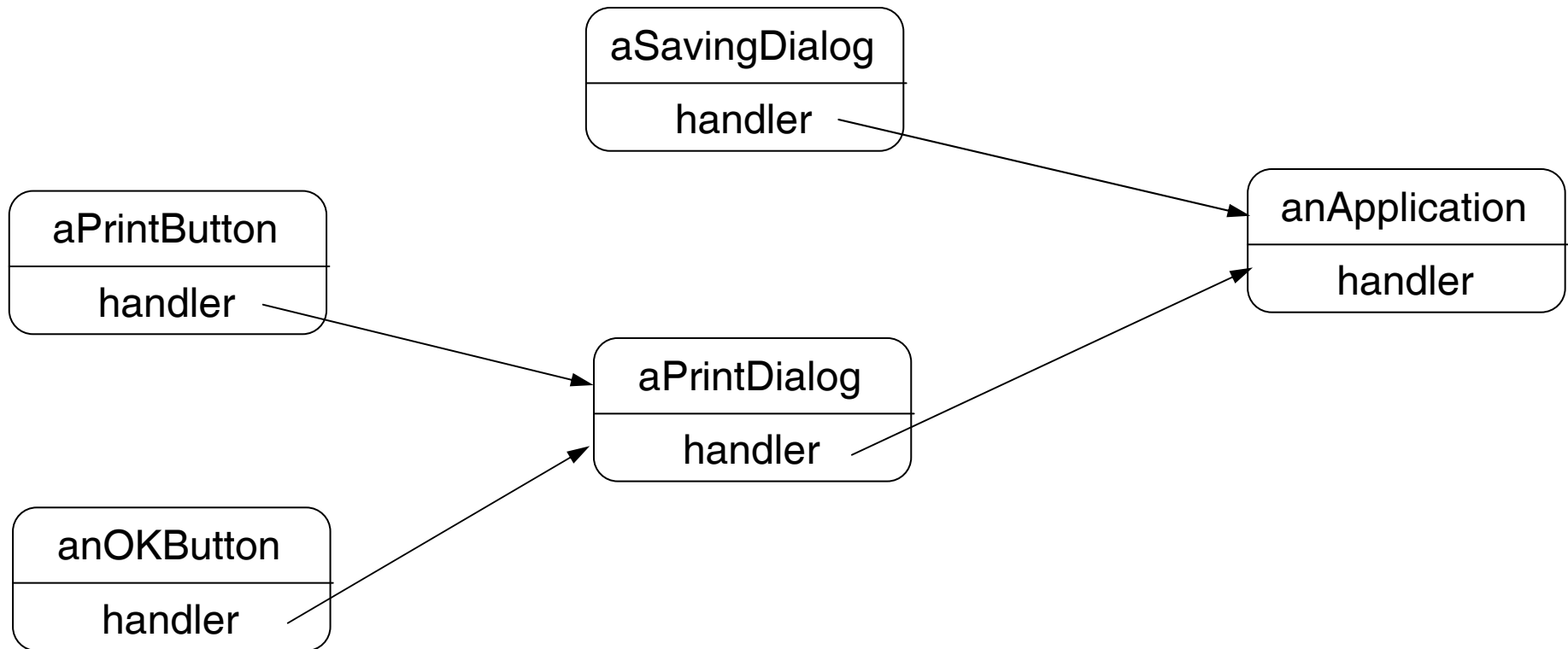


Mẫu Chain of Responsibility

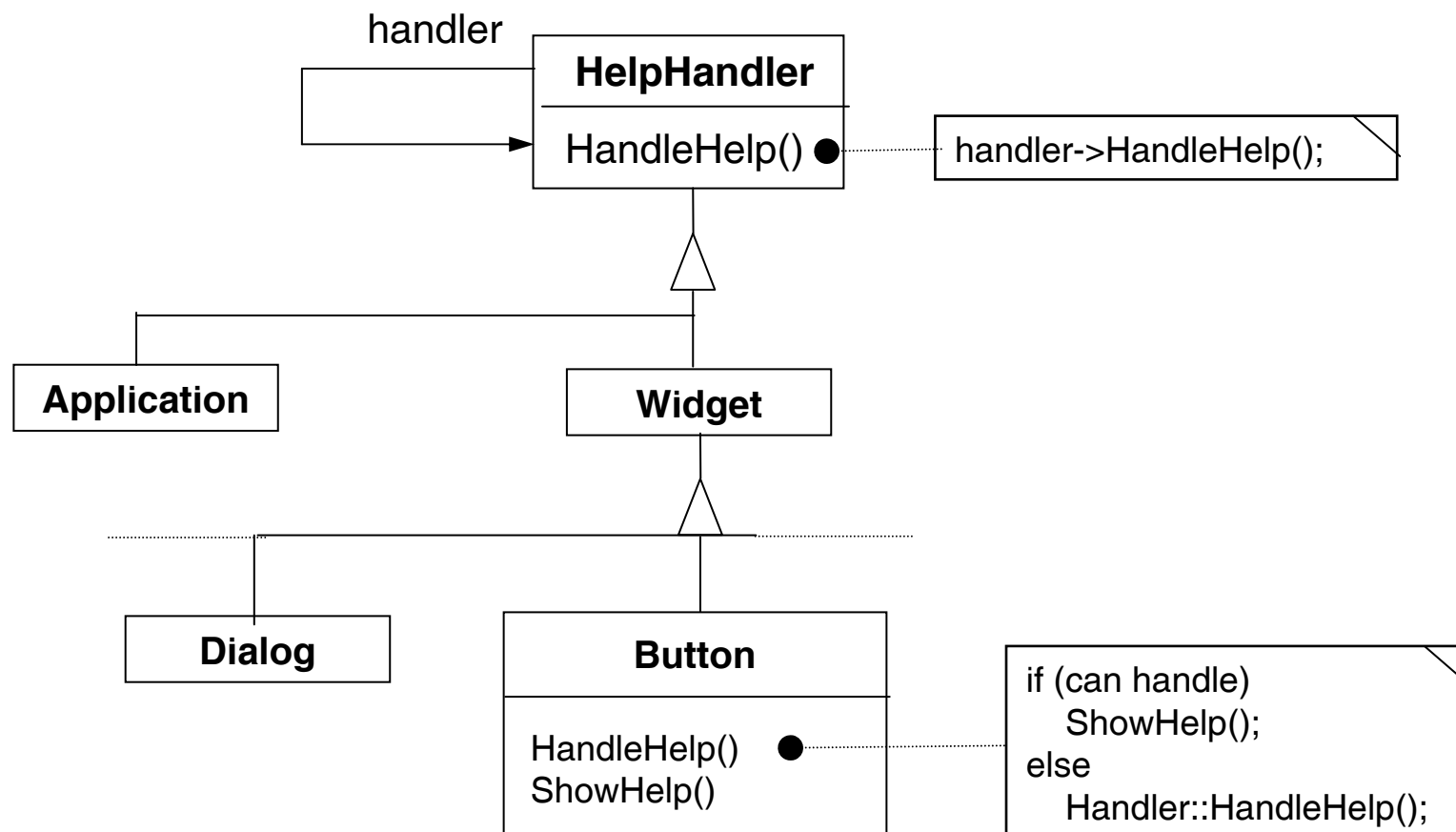
- ❑ Mục tiêu : Tránh việc gắn kết cứng giữa phần tử gửi request với phần tử nhận và xử lý request bằng cách cho phép hơn 1 đối tượng có cơ hội xử lý request. Liên kết các đối tượng nhận request thành dây chuyền rồi "pass" request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể.
- ❑ Nhu cầu áp dụng : Trong ứng dụng có trợ giúp theo ngữ cảnh thì user có thể thu được thông tin trợ giúp của 1 phần tử giao diện nào đó bằng cách chỉ cần click vào nó. Ta nên tổ chức thông tin trợ giúp theo tính tổng quát từ phần tử đặc biệt nhất (nhỏ nhất) đến tổng quát nhất (lớn nhất), mỗi thông tin trợ giúp được xử lý bởi đối tượng giao diện tương ứng phụ thuộc vào ngữ cảnh.

Thí dụ về mẫu Chain of Responsibility

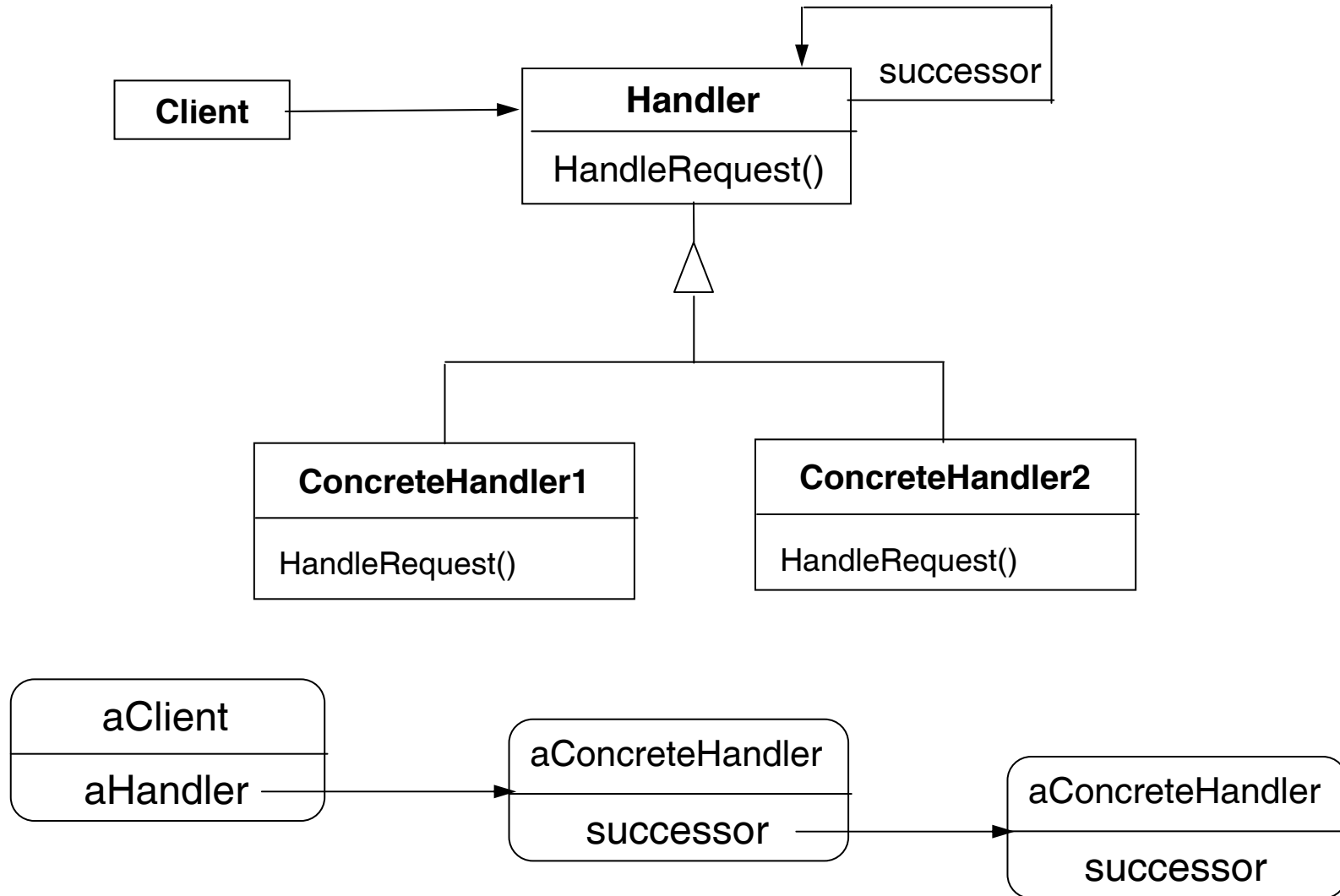
- Thí dụ khi ấn phải chuột vào button OK thì trình trợ giúp của OKButton sẽ chạy, nó sẽ hoặc hiển thị Help hoặc chuyển điều khiển cho trình trợ giúp của Dialog, trình trợ giúp của Dialog có thể chuyển điều khiển đến Application...



Thí dụ về mẫu Chain of Responsibility



Lược đồ cấu trúc của mẫu Chain of Responsibility



Các phần tử tham gia

❑ **Handler (HelpHandler) :**

- định nghĩa interface của tác vụ xử lý request.
- (optional) hiện thực mối liên kết đến đối tượng đi sau (successor).

❑ **ConcreteHandler (PrintButton, PrintDialog) :**

- xử lý request mà nó có trách nhiệm xử lý.
- có thể truy xuất đối tượng đi sau.
- nếu có thể xử lý được request, nó sẽ xử lý, nếu không forward request cho đối tượng đi sau giải quyết.

❑ **Client :**

- khởi động request và gọi tới 1 ConcreteHandler đầu tiên trong dây chuyền.

Các ngữ cảnh nên dùng mẫu Chain of Responsibility

- ❑ Thường áp dụng mẫu Chain of Responsibility trong các trường hợp sau:
 - hơn 1 đối tượng có thể xử lý request nhưng đối tượng nào sẽ xử lý thì chưa biết trước. Đối tượng xử lý sẽ được xác định động.
 - bạn muốn gửi request đến 1 đối tượng xử lý nào đó nhưng không xác định rõ ràng.
 - muốn xác định tập các đối tượng xử lý 1 request nào đó 1 cách động.



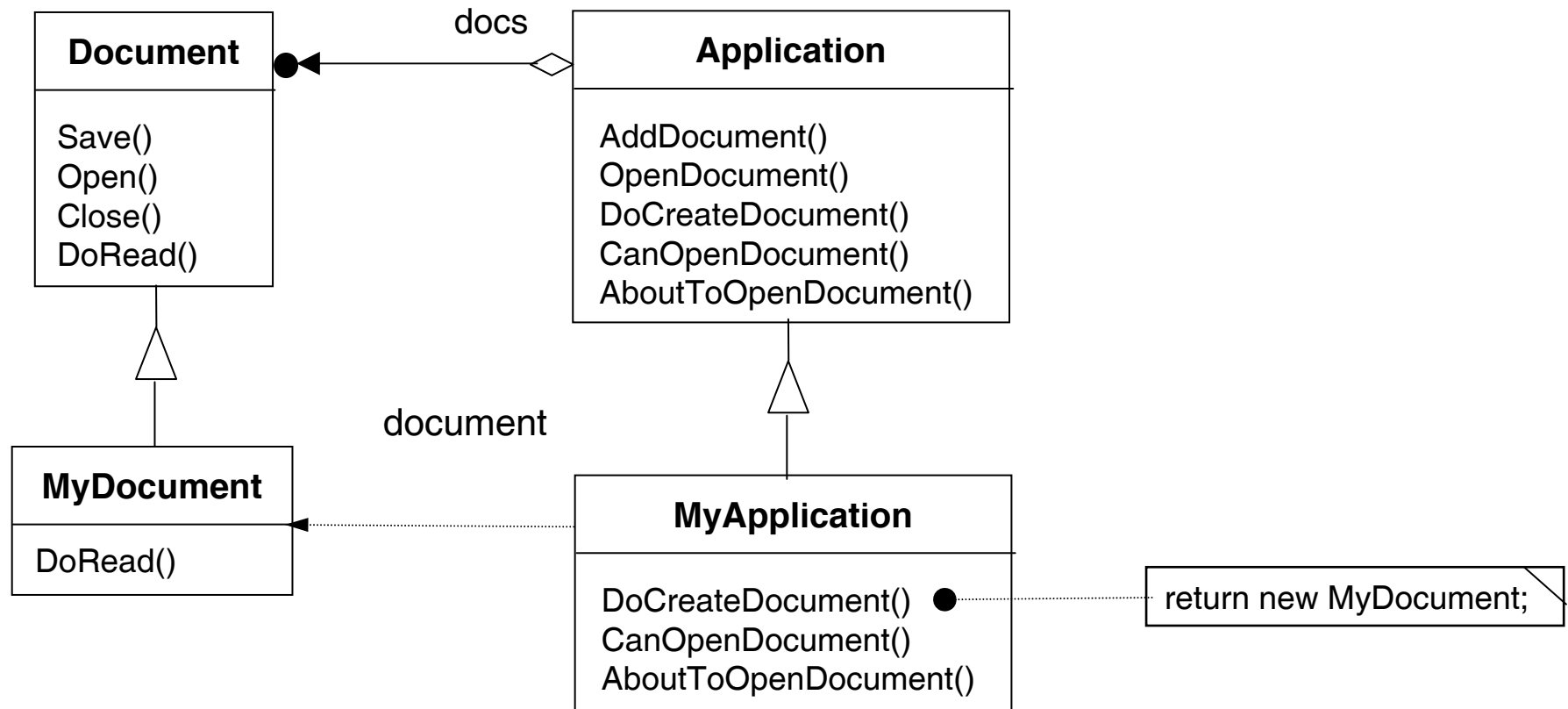
Template Method

- ❑ Mục tiêu : định nghĩa bộ khung giải thuật trong một tác vụ nhưng cho phép các class con hiện thực một số phần của tác vụ đó.
- ❑ Nhu cầu áp dụng : trong trường hợp Windows cho phép người lập trình Hook vào hệ thống, Windows đã để sẵn những entry mà người lập trình có thể chèn thêm phần xử lý của mình. Người lập trình không thể thay thế trình tự quá trình xử lý của Windows. Phương pháp lập trình hướng đối tượng có thể cung cấp hướng giải quyết những vấn đề này như sau:
 - Một lớp định nghĩa quá trình xử lý bao gồm nhiều tác vụ nhỏ hơn, các tác vụ nhỏ hơn có thể cho lớp con override chính là các điểm hook.
 - Các lớp con thừa kế hook vào quá trình xử lý của lớp cha bằng cách override các điểm hook.

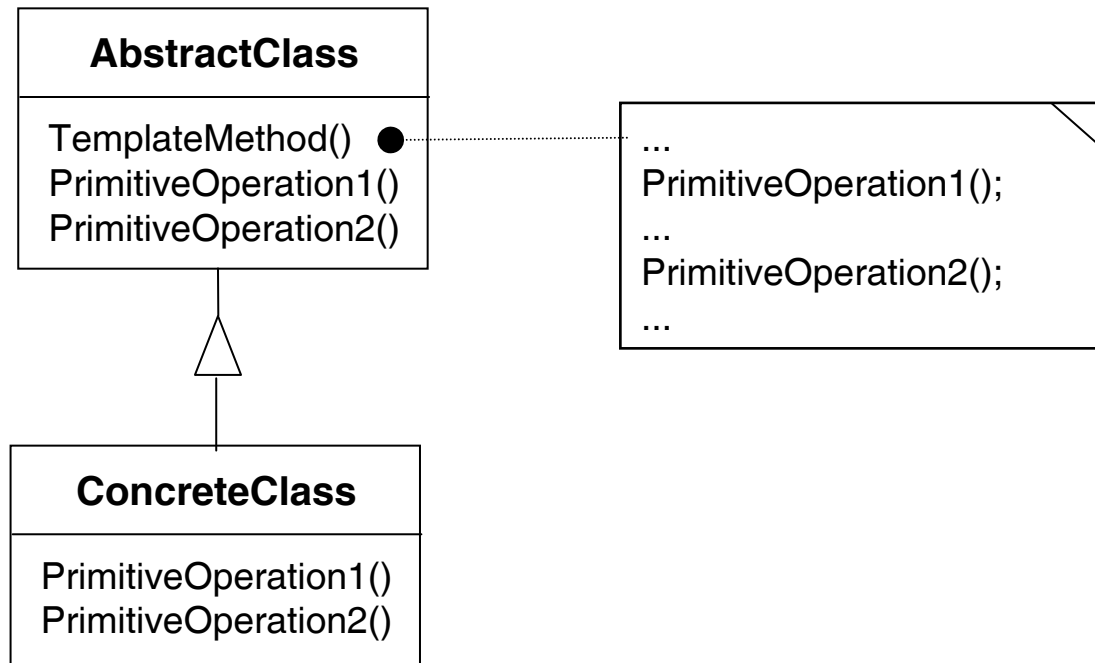
Hướng tiếp cận như trên là của mẫu Template Method.



Ví dụ về mẫu Template Method



Sơ đồ cấu trúc mẫu Template Method



Các phần tử tham gia

❑ **AbstractClass (Application) :**

- định nghĩa các primitive operation cho lớp con override để hiện thực một phần của hoạt động. Các phương thức này là điểm mà lớp con có thể hook vào code của lớp cha.
- hiện thực template method, là method định nghĩa bộ khung của hoạt động. Template method kết hợp các primitive operation để thực hiện trọn vẹn hoạt động.

❑ **ConcreteClass (MyApplication) :** hiện thực các primitive operation để can thiệp một phần vào quá trình thực hiện hoạt động ở lớp cha.



Các ngữ cảnh nên dùng mẫu Template Method

- ❑ Thường sử dụng Template method khi cần:
 - hiện thực một phần cố định của hoạt động và cho phép lớp con hiện thực phần có thể thay đổi.
 - tập trung các hành vi giống nhau ở các lớp để tránh trùng lặp.
 - kiểm soát quá trình override của lớp con: chỉ cho phép override những điểm hook qui định sẵn.



Mẫu Strategy

- ❑ Mục tiêu : Cung cấp một họ giải thuật và cho phép Client chọn lựa linh động một giải thuật cụ thể khi sử dụng.
- ❑ Nhu cầu áp dụng : Một số chương trình có nhiều giải thuật khác nhau cho cùng một vấn đề. Nhu cầu phát sinh: quản lý các giải thuật đó một cách đơn giản và cho phép client chọn một trong những giải thuật đó để sử dụng một cách linh động.



Ví dụ về nhu cầu ứng dụng Strategy

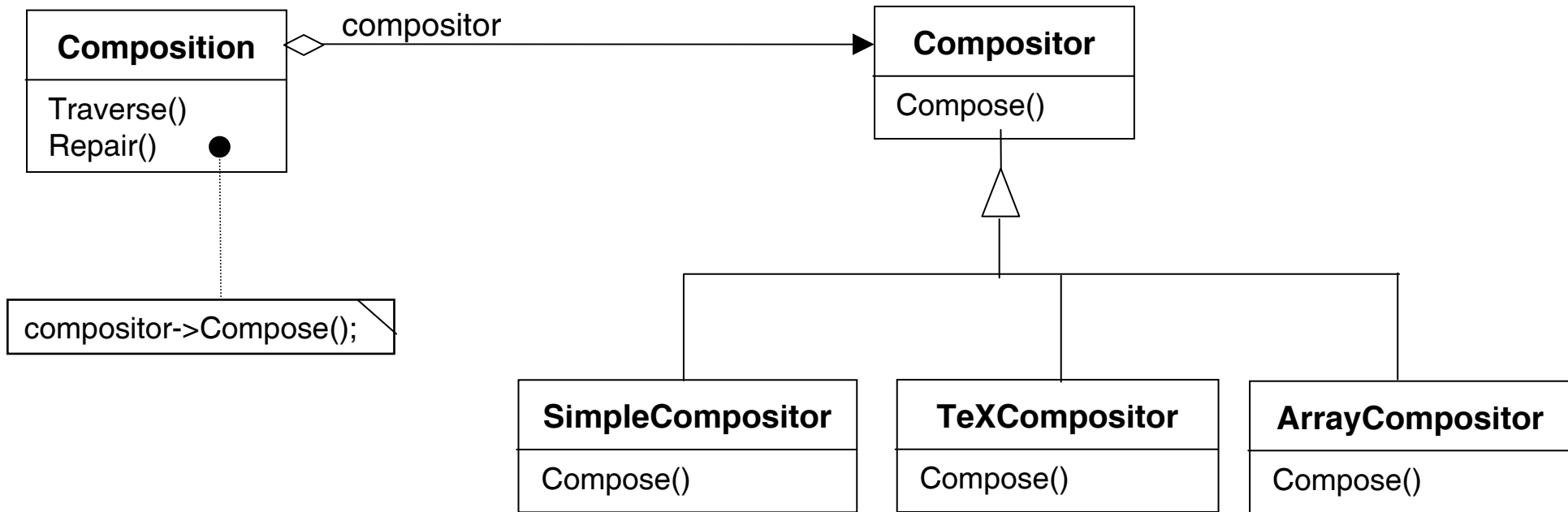
Chương trình chơi game có thể có nhiều giải thuật tùy vào mức độ khó của cuộc chơi. Khi người chơi chọn mức độ khó dễ chính là thao tác chọn giải thuật. Do đó đối tượng giải thuật phải tách biệt với code chương trình. Một hướng giải quyết được đề nghị như sau:

- Định nghĩa 1 interface chung cho các lớp thể hiện các giải thuật.
- Định nghĩa các lớp concrete hiện thực interface trên, mỗi lớp concrete thể hiện một giải thuật.
- Chương trình sử dụng đối tượng kiểu interface và cho phép client thay thế bằng đối tượng thể hiện giải thuật cụ thể khi chạy.

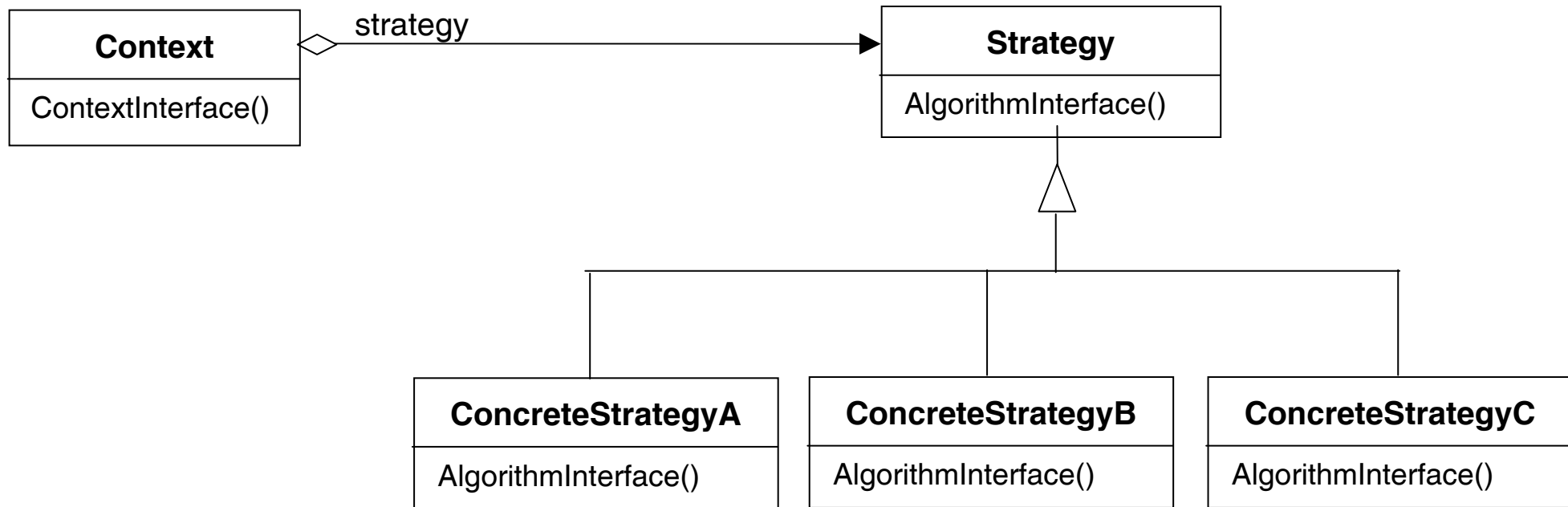
→ hướng giải quyết vấn đề của mẫu Strategy.



Ví dụ về mẫu Strategy



Lược đồ cấu trúc của mẫu Strategy



Các phần tử tham gia

- ❑ **Strategy (Compositor)** : định nghĩa interface cho tất cả các lớp thể hiện giải thuật. Có thể nhận pointer đến đối tượng Context trong quá trình khởi tạo đối tượng để truy xuất dữ liệu trong Context.
- ❑ **ConcreteStrategy (SimpleCompositor, TeXCompositor..)** : hiện thực interface Strategy, thể hiện một giải thuật cụ thể.
- ❑ **Context (Composition)** :
 - tại thời điểm dịch: chỉ sử dụng đối tượng kiểu Strategy khi xác định giải thuật cho vấn đề cần xử lý.
 - tại thời điểm run-time: được cung cấp một đối tượng giải thuật cụ thể thay thế cho đối tượng Strategy.
 - có thể cung cấp entry cho phép đối tượng kiểu Strategy truy xuất dữ liệu.

Các ngữ cảnh nên dùng mẫu Strategy

- ❑ Thường áp dụng mẫu Strategy trong các trường hợp sau:
 - Một lớp có nhiều hành vi loại loại trừ lẫn nhau và quá trình chuyển từ hành vi này sang hành vi khác cần được thực hiện dễ dàng. Khi đó mỗi hành vi sẽ được thể hiện trong 1 lớp Concrete Strategy và lớp có nhiều hành vi là lớp Strategy.
 - Giải thuật cần được che dấu cả về dữ liệu và cấu trúc đối với chương trình Client.
- ❑ Một số chương trình có thể áp dụng Strategy :
 - Compiler, OS: quá trình tối ưu hóa
 - Game: Quá trình chọn giải thuật
 - Các giao diện tổng quát (common dialog trong VB...)

Mẫu Command

- ❑ Mục tiêu : đóng gói request vào trong một Object, nhờ đó có thể thông số hóa chương trình nhận request và thực hiện các thao tác trên request: sắp xếp, log, undo...
- ❑ Nhu cầu áp dụng : Đôi khi chúng ta cần gửi request đến đối tượng nhưng không biết được hành động sẽ được thực thi cũng như những đối tượng bị tác động bởi request đó. Ví dụ user interface toolkit cần xây dựng trước việc truyền request đến menu và button nhưng chỉ có ứng dụng cụ thể mới xác định được hành động khi click vào chúng. Vấn đề này có thể được giải quyết như sau:

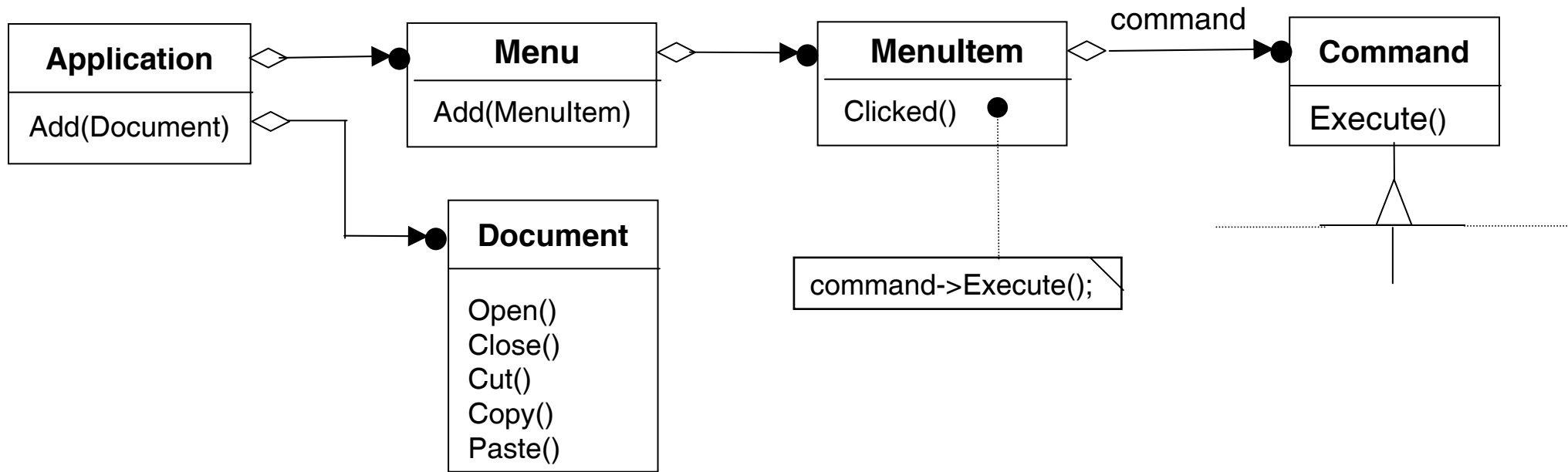


Ví dụ về nhu cầu ứng dụng mẫu Command

- Xây dựng lớp trừu tượng Command, trong đó có phương thức trừu tượng Execute().
- Menu hay button giữ liên kết đến đối tượng kiểu Command
- Request được đóng gói trong các đối tượng thừa kế Command. Các đối tượng này override phương thức Execute() để xác định hành động khi thực thi request.
- Khi ứng dụng cần gửi request đến cho menu hay button chỉ cần gửi đối tượng có đóng gói request đó đi. Menu hay button sẽ gọi phương thức Execute() trên đối tượng đó.

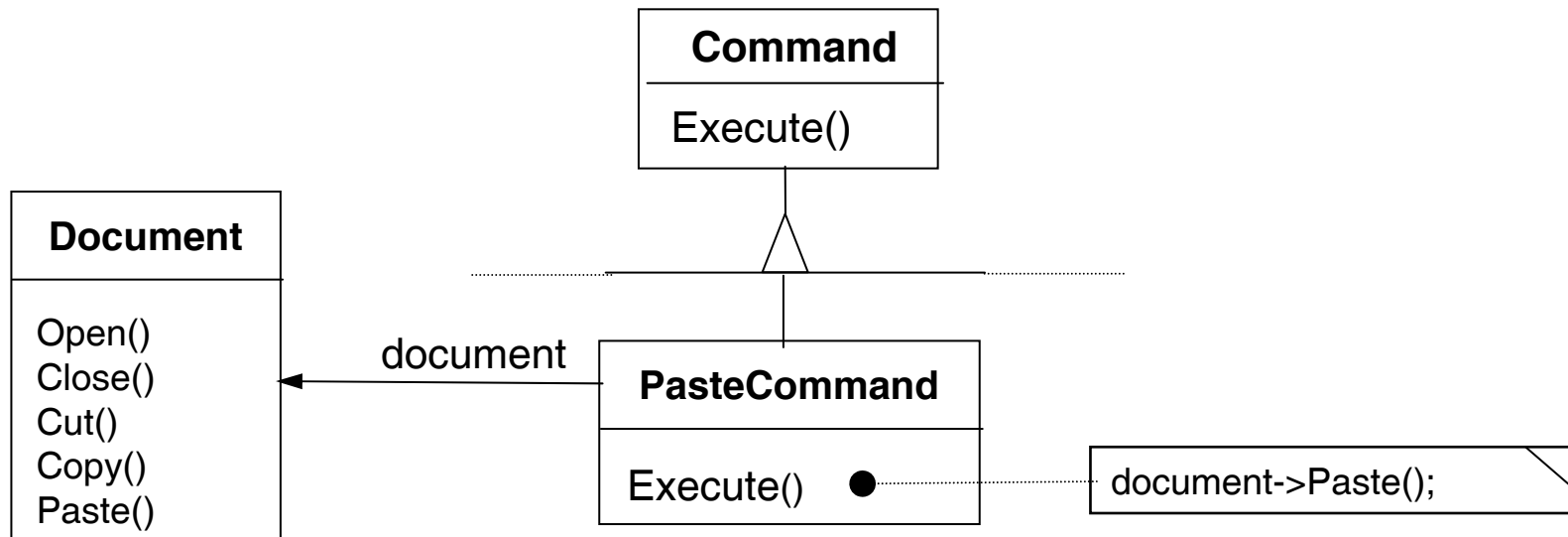


Ví dụ về mẫu Command



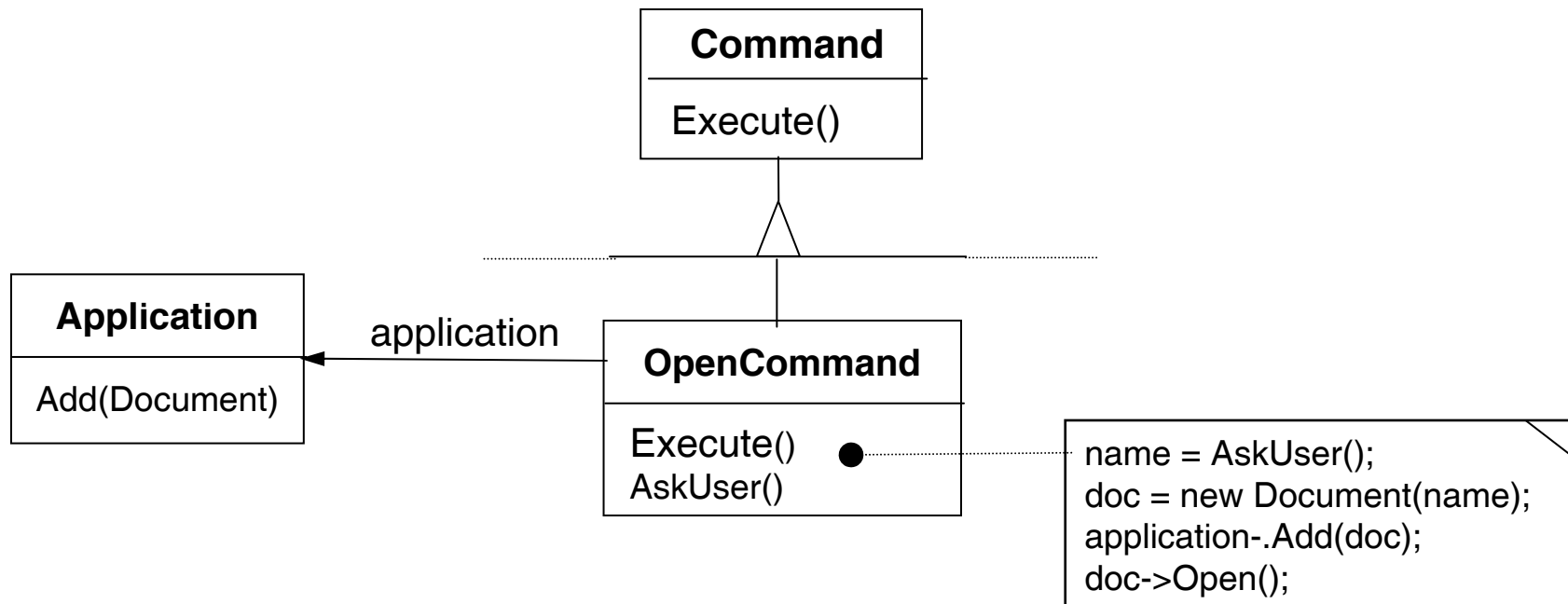
Ví dụ về mẫu Command

- Thí dụ đối tượng PasteCommand hỗ trợ hoạt động dán text từ clipboard vào 1 tài liệu. Phần tử nhận của PasteCommand là đối tượng Document mà PasteCommand được cung cấp trong lúc "instantiation". tác vụ Execute gọi chức năng Paste trên Document nhận được.



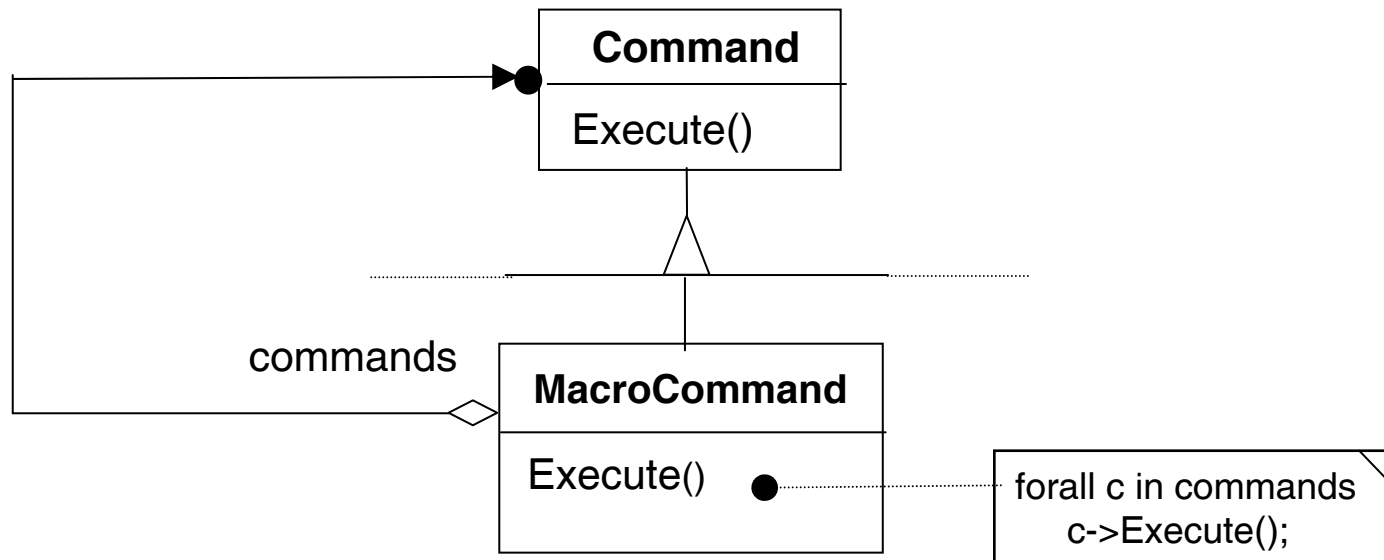
Ví dụ về mẫu Command

- ❑ Tác vụ Execute của OpenCommand thì khác : nó hiển thị cửa sổ yêu cầu user nhập tên document rồi tạo đối tượng Document tương ứng, "add" document vào ứng dụng nhận rồi mở document.

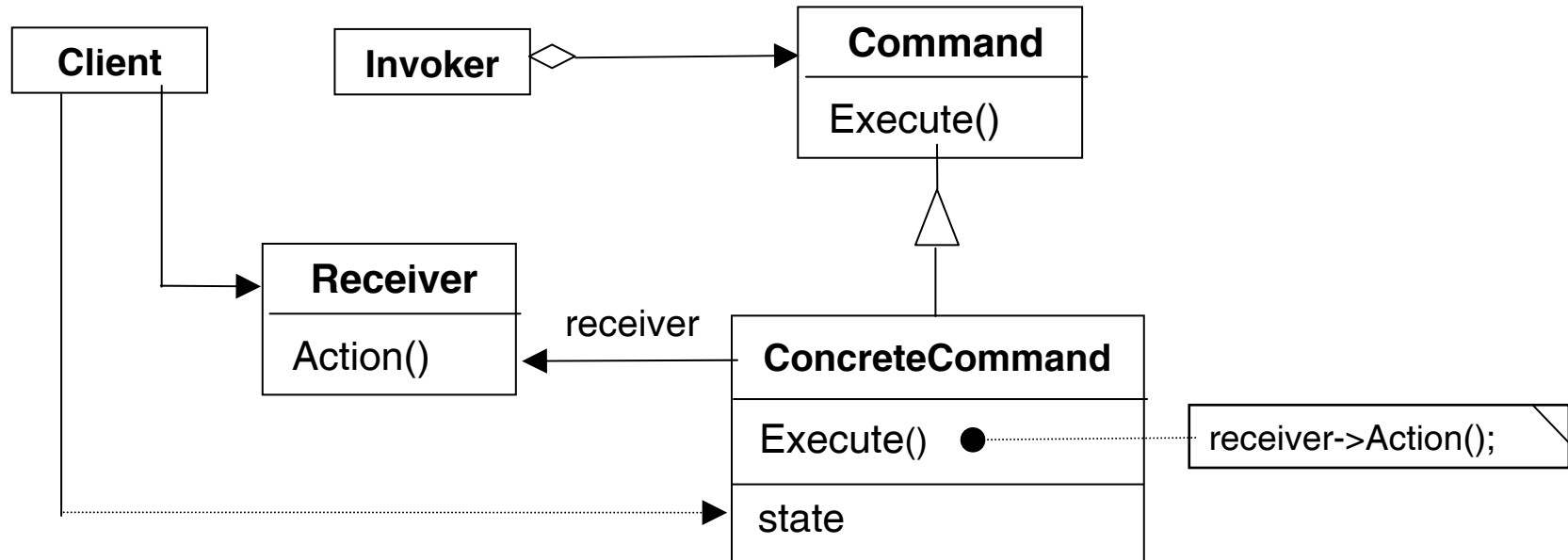


Ví dụ về mẫu Command

- Đôi khi 1 option Menu cần thực thi 1 chuỗi các lệnh, chúng ta có thể định nghĩa class MacroCommand để cho phép thi hành 1 số lệnh chưa biết trước.



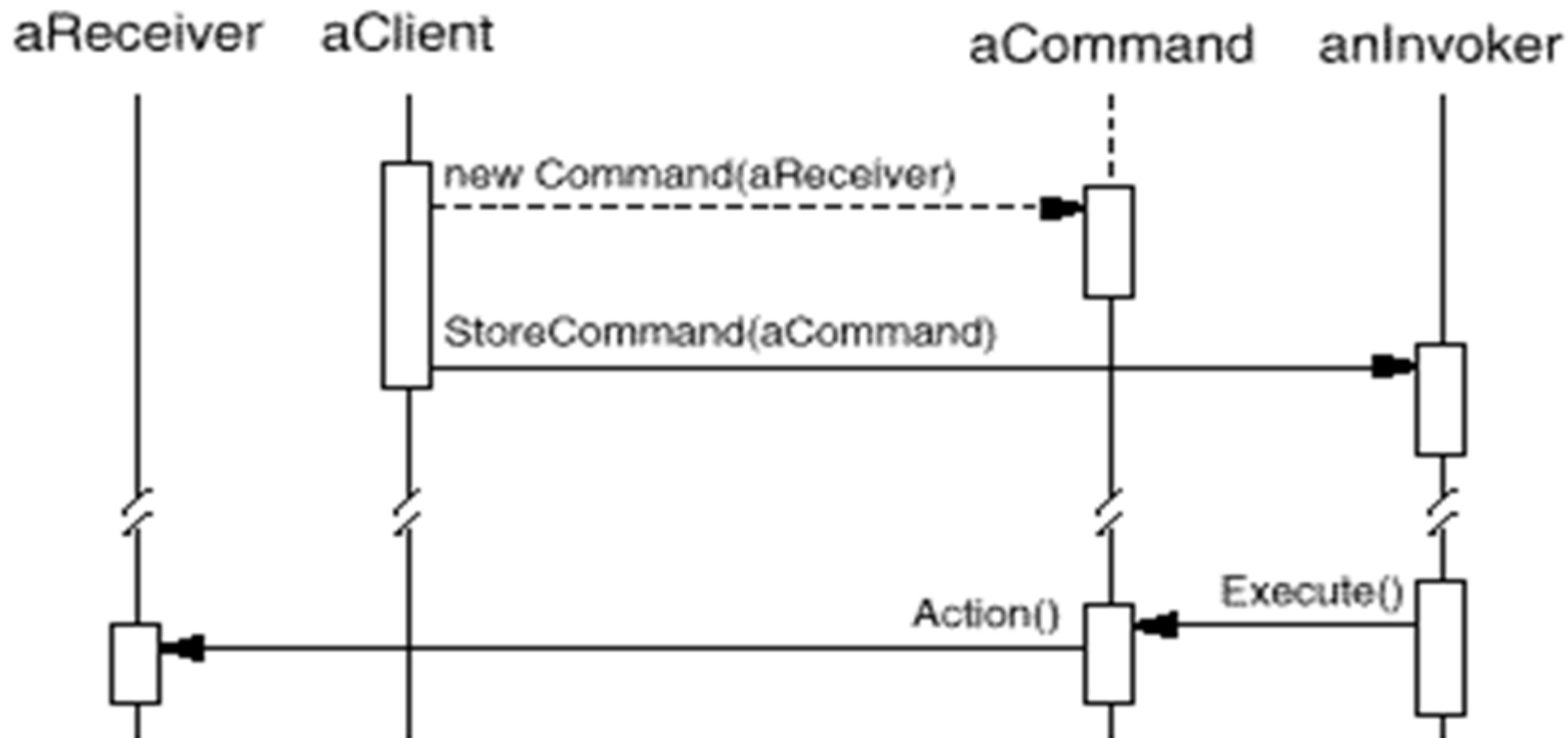
Lược đồ class của mẫu Command



Các phần tử tham gia

- ❑ **Command** : Khai báo các phương thức ảo (Execute()...) để gọi thực thi hay quản lý request.
- ❑ **ConcreteCommand (PasteCommand, OpenCommand):**
 - Xác định đối tượng nhận tương tác (đối tượng lớp Receiver).
 - Override phương thức Execute trong lớp Command để đáp ứng request.
- ❑ **Client (Application)** : khởi tạo đối tượng ConcreteCommand và truyền đối tượng nhận tương tác cho nó.
- ❑ **Invoker (MenuItem):** gửi request đến đối tượng Command.
- ❑ **Receiver (Document, Application)** :
 - Đối tượng nhận tương tác trong ConcreteCommand.
 - Biết cách thực hiện những hành động để đáp ứng request.

Quá trình cộng tác giữa các phần tử



Các ngữ cảnh nên dùng mẫu Command

- ❑ Thông số hóa đối tượng bằng hành vi mà đối tượng đó thực thi. Nghĩa là một entry của đối tượng có thể thực thi nhiều hành vi khác nhau tùy thuộc vào thông số (là đối tượng khác) truyền cho nó.
- ❑ Quản lý, lưu trữ và thực thi request tại những thời điểm khác nhau. Vì trong mẫu Command, request được lưu trữ trong các đối tượng nên việc lưu trữ và quản lý request chỉ là việc lưu trữ và quản lý các đối tượng.
- ❑ Hỗ trợ undo, redo các thao tác. Phương thức Execute() có thể lưu trạng thái cũ của đối tượng Receiver để phục hồi lại khi có yêu cầu.
- ❑ Hỗ trợ việc log lại các thay đổi trên đối tượng Receiver để có thể thực hiện trở lại trong trường hợp ứng dụng chưa lưu đối tượng Receiver đã thay đổi mà hệ thống lại bị hỏng. (Ví dụ MS Word có chức năng recovery).
- ❑ Các hệ thống hỗ trợ transaction. Trường hợp này có thể kết hợp mẫu Template Method.

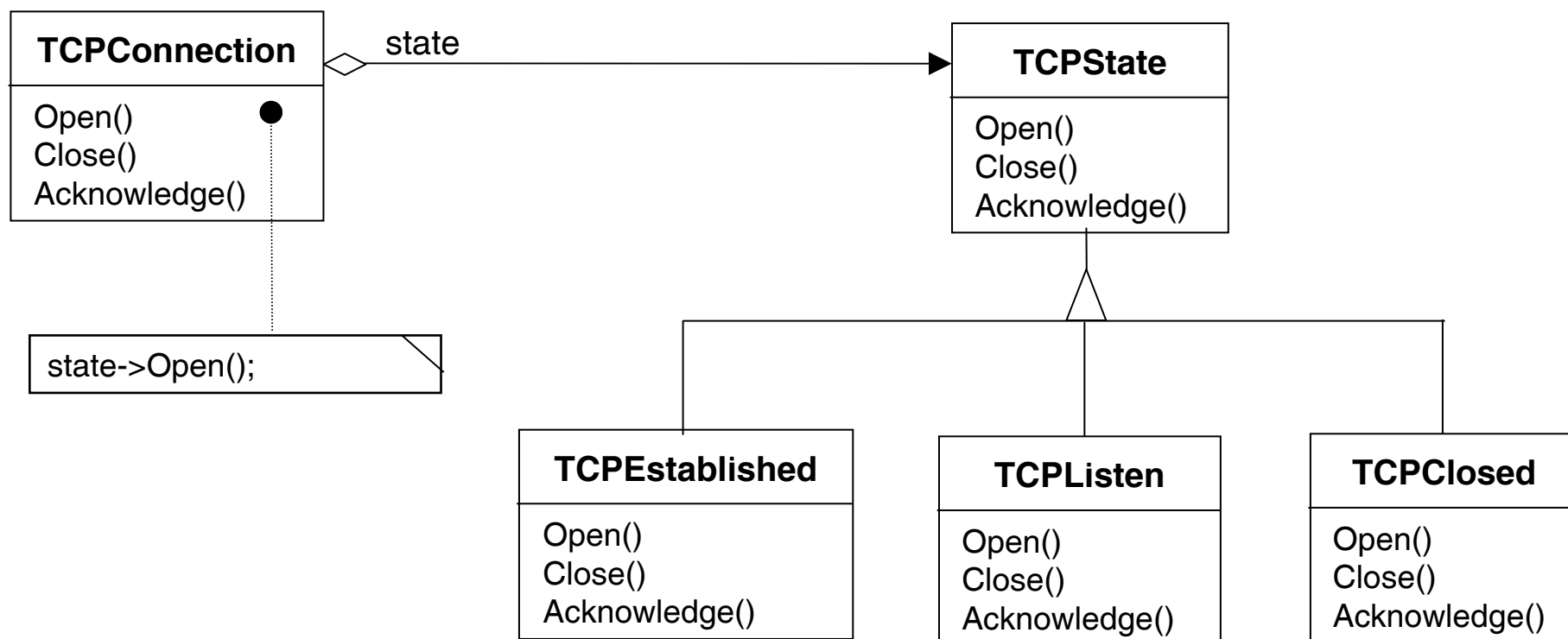
Mẫu State

- ❑ Mục tiêu : cho phép 1 đối tượng thay đổi hành vi khi trạng thái bên trong của nó thay đổi. Ta có cảm giác như class của đối tượng bị thay đổi.
- ❑ Nhu cầu áp dụng : trong class TCPConnection miêu tả 1 mối nối mạng, đối tượng TCPConnection có thể ở 1 trong nhiều trạng thái : Established, Listening, Closed. Khi đối tượng TCPConnection nhận request, nó sẽ đáp ứng khác nhau tùy vào trạng thái hiện hành.

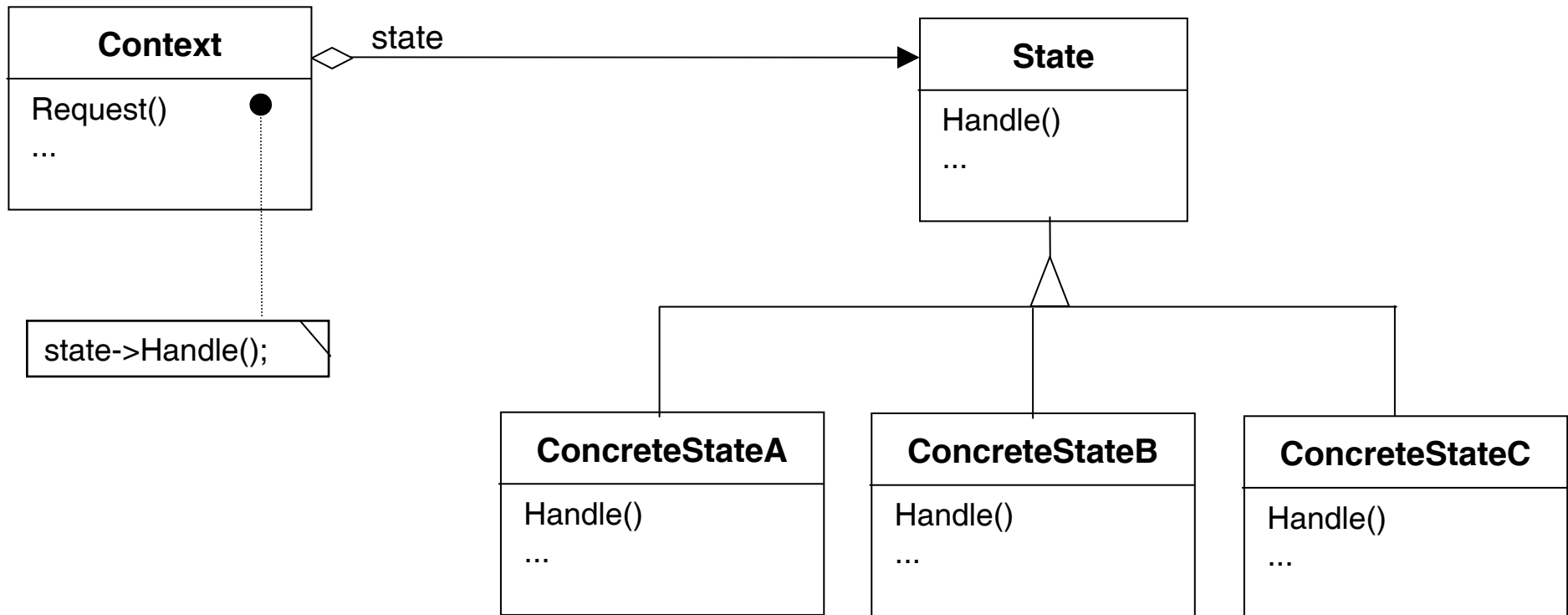
→ hướng giải quyết vấn đề của mẫu State.



Ví dụ về mẫu State



Lược đồ cấu trúc của mẫu State



Các phần tử tham gia

❑ **Context (TCPConnection) :**

- định nghĩa interface cần dùng cho client.
- duy trì 1 tham khảo đến đối tượng của 1 class con ConcreteState mà định nghĩa trạng thái hiện hành.

❑ **State (TCPState) :**

- định nghĩa interface nhằm bao đóng hành vi kết hợp với trạng thái cụ thể.
- duy trì 1 tham khảo đến đối tượng của 1 class con ConcreteState mà định nghĩa trạng thái hiện hành.

❑ **ConcreteState (TCPEstablished, TCPListen, TCPClose) :**

- định nghĩa hành vi cụ thể kết hợp với trạng thái của mình.

Các ngữ cảnh nên dùng mẫu State

- ❑ Thường áp dụng mẫu State trong các trường hợp sau:
 - hành vi của đối tượng phụ thuộc vào trạng thái của nó và phải thay đổi run-time khi trạng thái thay đổi.
 - các tác vụ có những lệnh điều kiện số học lớn phụ thuộc vào trạng thái đối tượng.



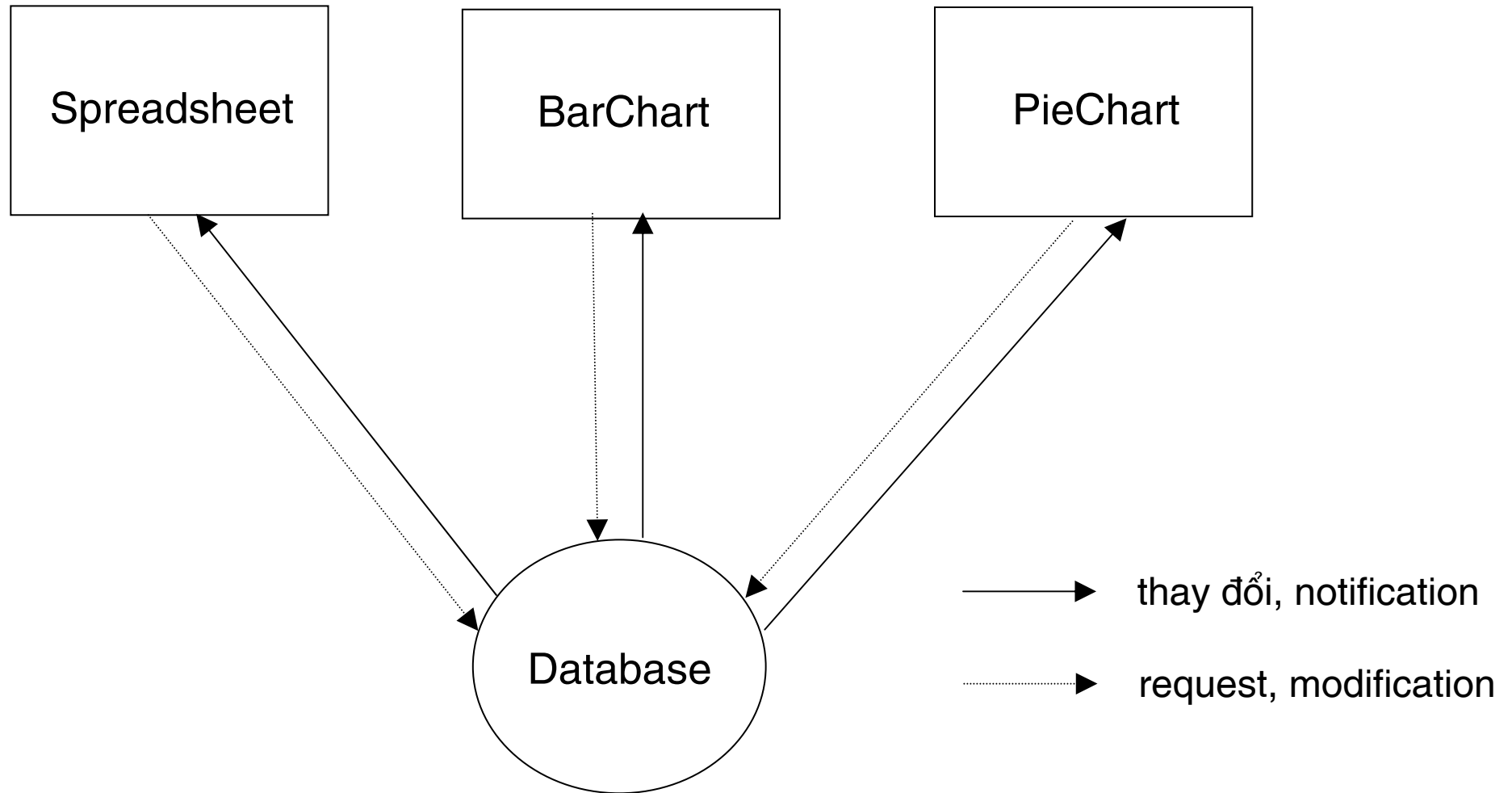
Mẫu Observer

- ❑ Mục tiêu : định nghĩa sự phụ thuộc 1-n giữa các đối tượng sao cho khi 1 đối tượng thay đổi trạng thái thì các đối tượng phụ thuộc được cảnh báo hiệu chỉnh tự động (để đảm bảo tính nhất quán).
- ❑ Nhu cầu áp dụng : trong ứng dụng quản lý bảng tính, mỗi bảng tính là 1 database nhưng nó được trình bày dưới nhiều dạng khác nhau như spreadsheet, barchart, piechart,... Mỗi khi nội dung database thay đổi ta muốn cập nhật đồng thời nhiều dạng biểu diễn nó.

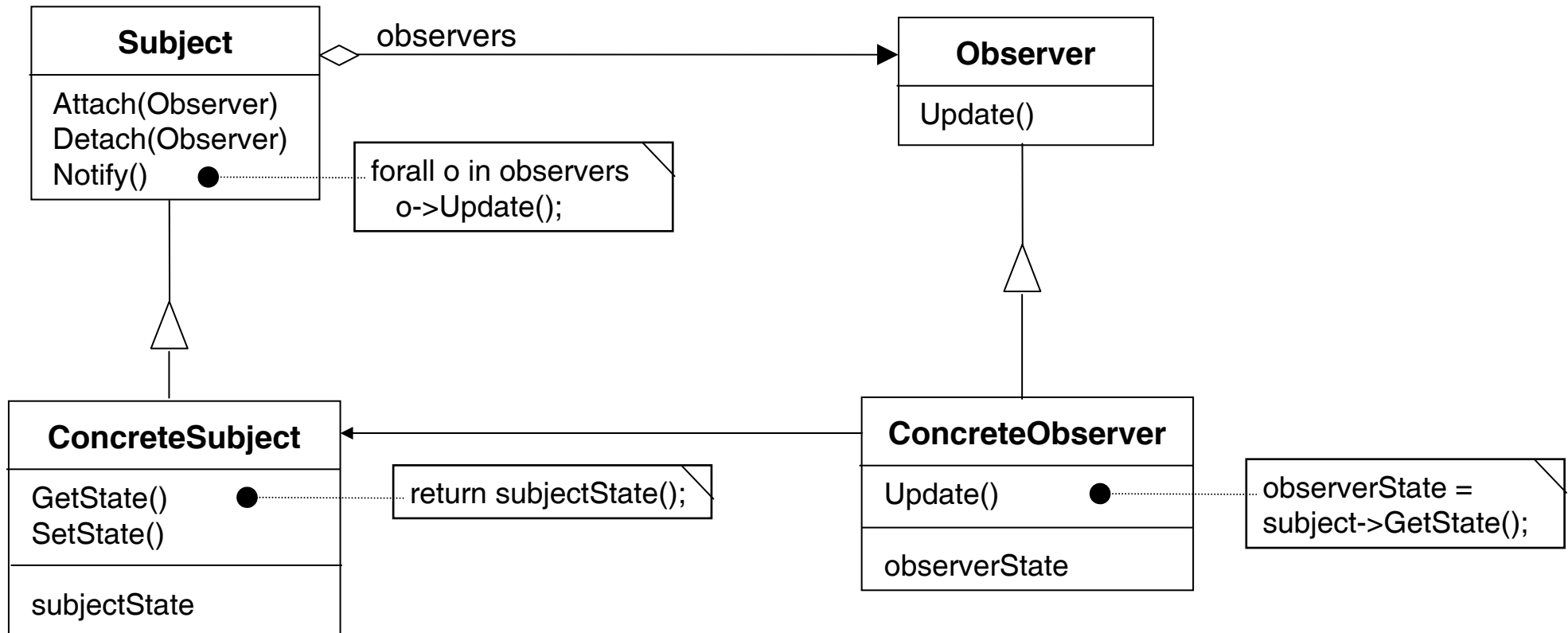
→ hướng giải quyết vấn đề của mẫu Observer.



Ví dụ về mẫu Observer



Lược đồ cấu trúc của mẫu Observer



Các phần tử tham gia

❑ **Subject :**

- biết observer của nó. Có thể có nhiều observer quan sát 1 subject.
- cung cấp interface để Attach và Detach các observer vào mình.

❑ **Observer :**

- định nghĩa interface hiệu chỉnh cho các đối tượng mà sẽ được cảnh báo để hiệu chỉnh subject của mình.

❑ **ConcreteSubject :**

- lưu trạng thái lưu ý tới các đối tượng ConcreteObserver.
- gửi cảnh báo tới các observer khi trạng thái của nó thay đổi.

❑ **Concretebserver :**

- duy trì tham khảo tới đối tượng ConcreteSubject.
- lưu trạng thái mà cần phải luôn nhất quán với subject của mình.
- hiện thực interface hiệu chỉnh để giữ trạng thái luôn nhất quán với subject của mình.

Các ngữ cảnh nên dùng mẫu Observer

- ❑ Thường áp dụng mẫu State trong các trường hợp sau:
 - khi 1 sự trừu tượng có 2 khía cạnh phụ thuộc lẫn nhau. Bao đóng các khía cạnh này trong những đối tượng độc lập giúp ta thay đổi và dùng lại chúng độc lập.
 - khi việc thay đổi đối tượng này đòi hỏi phải thay đổi các đối tượng khác nhưng bạn không biết trước có bao nhiêu đối tượng cần thay đổi.
 - khi đối tượng cần có khả năng cảnh báo các đối tượng khác nhưng không muốn chúng ghép nối chặt với nhau.

