EC3307:Object & Component Technology

Design By Contract (2)

Defensive Programming vs. Design by Contract

- Defensive programming is an approach that promotes putting checks in every module to detect unexpected situations

- This results in redundant checks (for example, both caller and callee may check the same condition)
  - A lot of checks make the software more complex and harder to maintain

- In Design by Contract the responsibility assignment is clear and it is part of the module interface
  - prevents redundant checks
  - easier to maintain
  - provides a (partial) specification of functionality

Defensive Programming vs. Design by Contract

- Design by Contract is complementary to defensive programming because
  - With preconditions, it makes clear which inputs (to methods) are unexpected.
  - With postconditions, it makes it clear when an internal bug has occurred.
- But it does not prescribe predictable behaviour in the face or unexpected inputs and internal errors.

Defensive programming and contracts

- A contract obviously guides the writing of run-time defensive checks.
  - Of course if contracts can be proved to be respected, there is no need for defensive checks.
- Systems such as JML, Spec#, and .NET Contracts can automatically turn contracts into run-time defensive checks.
  - Systems such as JML, Spec#, and .NET Contracts can automatically verify that contracts are respected.

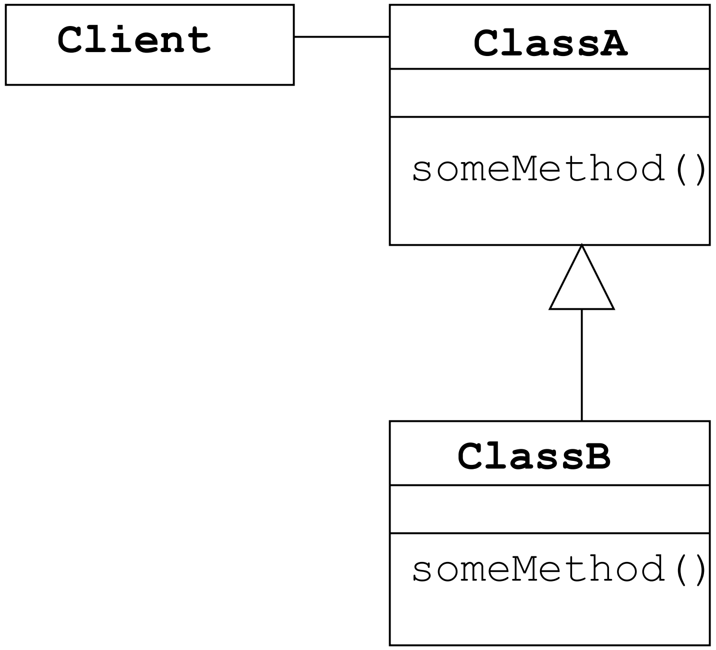
Design by Contract and Inheritance

- Inheritance enables declaration of subclasses which can redeclare some of the methods of the parent class or provide an implementation for the abstract methods of the parent class.
- Polymorphism and dynamic binding combined with inheritance are powerful programming tools provided by object-oriented languages
  - How can the Design by Contract can be extended to handle these concepts?

Inheritance: Preconditions

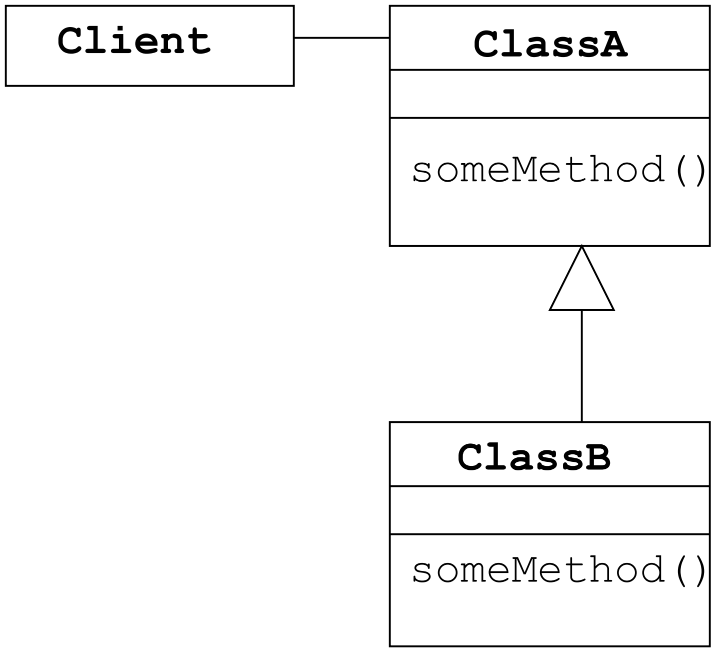
- If the precondition of the `ClassB.someMethod` is stronger than the precondition of the `ClassA.someMethod`, then this is not fair to the `Client`

- The code for `ClassB` may have been written after `Client` was written, so `Client` has no way of knowing its contractual requirements for `ClassB`



Inheritance: Postconditions

- If the postcondition of the `ClassB.someMethod` is weaker than the postcondition of the `ClassA.someMethod`, then this is not fair to the `Client`
- Since `Client` may not have known about `ClassB`, it could have relied on the stronger guarantees provided by the `ClassA.someMethod`



Inheritance

- Inherited contracts enforce the following
  - the precondition of a derived method to be weaker
  - the postcondition of a derived method to be stronger

- When a method overwrites another method, the new declared precondition is combined with previous precondition using disjunction

- When a method overwrites another method the new declared postcondition is combined with previous postcondition using conjunction

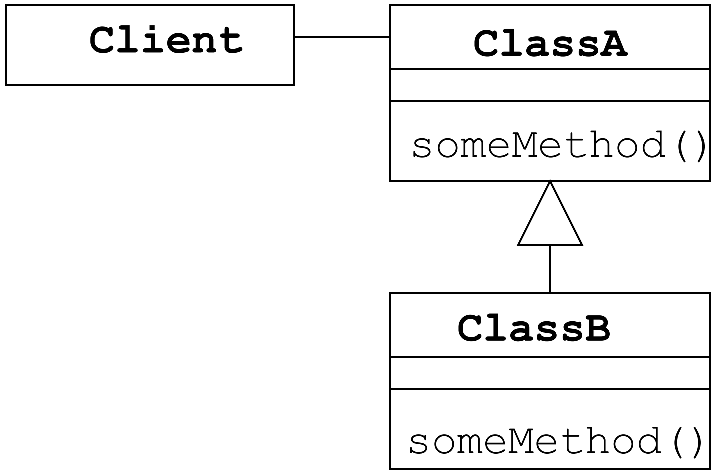
- Also, the invariants of the parent class are passed to the derived classes
  - invariants are combined using conjunction

In `ClassA`:

```
invariant
    classInvariant
someMethod() is
require
    Precondition
do
    Procedure body
ensure
    Postcondition
end
```

In `ClassB` which is derived from `ClassA`:

```
invariant
    newClassInvariant
someMethod() is
require
    newPrecondition
do
    Procedure body
ensure
    newPostcondition
end
```



The precondition of `ClassB.aMethod` is defined as:
`newPrecondition` **`or`** `Precondition`

The postcondition of `ClassB.aMethod` is defined as:
`newPostcondition` **`and`** `Postcondition`

The invariant of `ClassB` is
`classInvariant` **`and`** `newClassInvariant`

Liskov Substitution Principle (LSP)

An instance of a derived class should be able to replace any instance of its superclass.

→ Implement classes such that a derived class may be used any place that the base class may be used.

→ In effect, the base class establishes an interface and basic behaviour (functionality).

If a derived class violates this principle, then it has most likely violated the LSP.

- For example consider an `ellipse` base class extended by a `circle` derived class

- While a circle may be considered to be a special case of an ellipse, if you change the major axis values of the ellipse then it may not be a circle, etc.

Derived class methods (see “Design by contract”):

- a subclass should honour the “contracts” made by its parent class.

- pre-conditions should be no stronger than in the base class method.

-post-conditions should be no weaker than in the base class method.

Demand no more, promise no less

Demand no more: the subclass should accept any arguments that the superclass would accept.

Promise no less: Any assumption that is valid when the superclass is used must be valid when the subclass is used.

The Liskov Substitution Principle: Methods that use references to base classes must be able to use objects of derived classes without knowing it.

Example: If you have a variable of type `Animal`, the code that uses this variable should not care what kind of `Animal` it is.

```
Animal x; ...; if (x instanceof Frog) { ... }
```

If you introduce a `Deer` class, you should not have to make any changes to code that uses an `Animal`

```
… else if (x instanceof Deer) { ... } // new code
```

If you do have to change code, your `Animal` class was poorly designed

Example

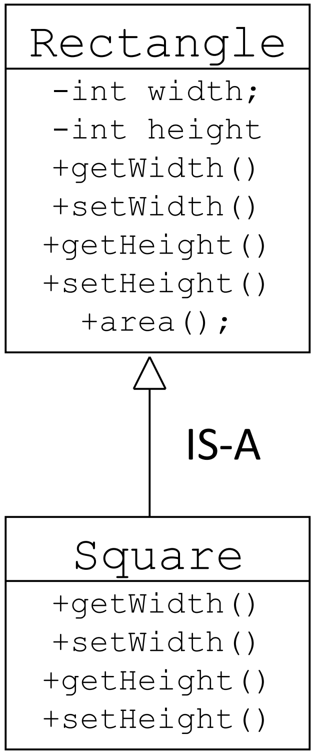
```
class Rectangle {
   public void setWidth(int width) {
       setWidth(width);
}
   public void setHeight(int height) {
       super.setHeight(height);
}
}

void clientOfRectangle(Rectangle r) {
    r.setWidth(10);
    r.setHeight(20);
    print(r.area());
}
```

Can we derive a Square class from the Rectangle class?

```
class Square extends Rectangle {
   public void setWidth(int width) {
       super.setWidth(width);
       super.setHeight(width);
   }
   public void setHeight(int height) {
       super.setHeight(height);
       super.setWidth(height);
   }
}
void clientOfRectangle(Rectangle r) {
    r.setWidth(10);
    r.setHeight(20);
    print(r.area());
}

Rectangle r = new Square(…);
clientOfRectangle(r); // what would be printed?
```



Is a square a rectangle?

Why is the Liskov Substitution Principle important?

1. Because if not followed, then class hierarchies will be a mess.

   Mess being that whenever a subclass instance was passed as parameter to any method, strange behaviour would occur.

2. Because if not followed, unit tests for the superclass would never succeed for the subclass.

Liskov Substitution Principle (LSP)

All derived classes must be substitutable
for their base class

Barbara Liskov, 1988

The "Design-by-Contract" formulation:

All derived classes must honour the contracts
of their base classes

Bertrand Meyer

Command Query Separation

- According to this principle, a method should either be:
- A **command** that performs an action and is void with no return value (or)
- A **query** that returns data to the caller and has no side effects
- A method can do either, but should not do both. If it does both, and you later try to use the side effect value, it is hard to determine where and when it was set. You might later want to access the value without changing it, and change it by accident.

Why CQS?

- Makes designs easier to understand
- No surprise side effects