# CS 619 Introduction to OO Design and Development

## Design by Contract

Fall 2013

## Design by contract

What is meant by "design by contract" or "programming by contract"?

**contract**: An agreement between classes/objects and their clients about how they will be used.

- used to assure that objects always have valid state
- non-software contracts: bank terms, product warning labels

- To ensure every object is valid, show that:
  - constructors create only valid objects
  - all mutators preserve validity of the object
- How to enforce a contract?



## Example contract issue

- A potential problem situation: queue class with `remove` or `dequeue` method
  - client may try to remove from an empty queue
- What are some options for how to handle this?
  - declare it as an error (an exception)
  - tolerate the error (return null)
  - print an error message inside the method
    - bad because it should leave this up to the caller
  - repair the error in some way (retry, etc.)
    - bad because it should leave this up to the caller

The decision we make here becomes part of the contract of the queue class!



## Design by Contract

- Proposed by Bertrand Meyer for Eiffel
- Organize communication between software elements by organizing mutual **obligations** and **benefits**
- Use a metaphor of
  - clients: receive services from suppliers
  - suppliers: supply services

| | Client | Supplier |
|---|---|---|
| **Obligation** | Satisfy supplier requirement | Guarantee service |
| **Benefit** | Get service | Impose requirements |

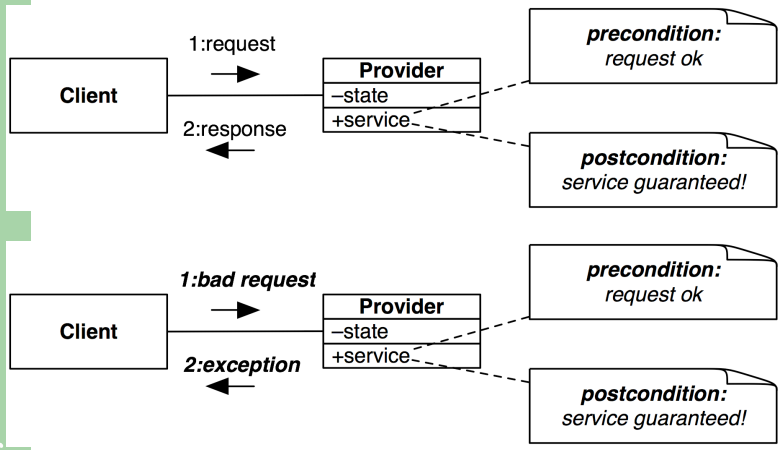
# Example:

| | Client | UPS |
|---|---|---|
| **Obligation** | Provide package of no more than 5kg, each dimension less than 2 meters. Pay postage | Deliver package to recipient in 24 hours or less |
| **Benefit** | Get package delivered in 24 hours or less | No need to deal with deliveries that are too big, too heavy or unpaid |

| | Client | Supplier |
|---|---|---|
| **Obligation** | Precondition | Post-condition |
| **Benefit** | Post-condition | Precondition |



# Design by Contract == Don't accept anybody else's garbage!






# Pre-condition

- What happens when a precondition is not met?

**precondition**: Something that must be true before object promises to do its work.

- Example: A hash map class has a put(*key*, *value*) and a get(*key*) method.
  - A precondition of the get method is that the key was not modified since the time you put it into the hash map.
- If precondition is violated, object may choose any action it likes
  - If key was modified, the hash map may state that the key/value is not found, even though it is in the map.
- Document preconditions in Javadoc with tag **`@pre.condition`**



# Post-conditions

- Whose fault is it when a postcondition is not met, and what should be done?

- **postcondition**: Something that must be true upon completion of the object's work.
  - Example: At end of sort(int[]), the array is in sorted order.
  - Check them with statements at end of methods, if needed.
  - A postcondition being violated is object's (your) own fault.
  - Assert the postcondition, so it crashes if not met.
    - Don't throw an exception -- it's not the client's fault!

- Document postconditions in Javadoc with tag **`@post.condition`**

## Class invariants

- How is it enforced?

**class invariant**: A logical condition that always holds for any object of a class.

- Example: Our account's balance should never be negative.
- Similar to loop invariants, which are statements that must be true on every iteration of a loop.
- Can be tested at start or end of every method.
- Assert your invariants, so it crashes if they are not met don't throw an exception -- it's not the client's fault!

- Document class invariants in the Javadoc comment header at the top of the class. (no special javaDoc tag)



## Programming Language Support

- Eiffel : Design as such, but limited usage
- C++:
  - assert() does not throw an exception
  - Documentation extraction difficult
- Java
  - ASSERT is standard since Java 1.4
  - JavaDoc annotation



## Effiel Example: Stack

```
class stack
    invariant: (isEmpty (this)) or
        (! isEmpty (this))

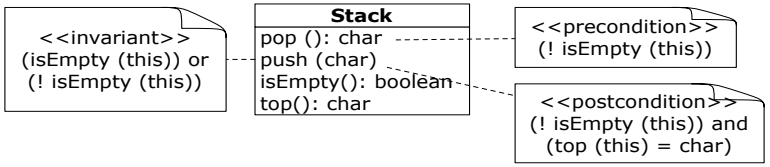
    public char pop ()
        require: ! isEmpty (this)
            ensure: true

    public void push (char)
        require: true
        ensure: (! isEmpty (this))
            and (top (this) = char)
```

Implementors of stack promise that invariant will be true after all methods return (incl. constructors)

Clients of stack promise precondition will be true before calling pop()

Implementors of stack promise postcondition will be true after push() returns






## Effiel Example: Map insert()

```
put (x: ELEMENT; key: STRING) is
                -- Insert x so that it will be retrievable through key.
        require
                count <= capacity
                not key.empty
        do
                ... Some insertion algorithm ...
        ensure
                has (x)
                item (key) = x
                count = old count + 1
        end
```

## Assertion Violations

The assertions can be monitored dynamically at run-time to debug the software

- A precondition violation would indicate a bug at the caller
- A postcondition violation would indicate a bug at the callee

Our goal is to prevent assertion violations from happening

- The pre and postconditions are not supposed to fail if the software is correct ( so they differ from exceptions and exception handling)



## Assertions in Java

- Java assert statements
  - `assert` ***<condition>*** `;`
  - `assert <condition> : <message>;`
- will raise an `AssertionError` if `<condition>` is false.
- enabling assertions
  - when compiling: `javac -source 1.5 ClassName.java`
  - when running: `java -ea ClassName`
- In C/C++, assert is a compile-time thing. In Java, can selectively en/disable assertions at runtime.



## Debug and ship builds

Most companies have at least two versions of their code:

- *debug build* : has special code only for the developer
  - debug print statements or graphical output
  - assertions for pre/postconditions, invariants
- *ship build* : meant to be used by customers
  - Users expect reasonable performance and reliability.
  - The app should not spend a lot of time checking for pre/post or invariants (in ship build).
- The same code is used to make debug and ship build.
  - special flags (e.g. DEBUG_MODE) turn on and off debug code
  - in Java, VM can be run with flags to turn on/off debug also



## Checking Pre-conditions

Assert pre-conditions to inform clients when *they* violate the contract.

```
public Object top() {
   assert(!this.isEmpty()); // pre-condition
   return top.item;
}
```

*Always check pre-conditions, raising exceptions if they fail.*

## Checking Class Invariants

Every class has its own invariant:

```
protected boolean invariant() {
   return (size >= 0) &&
      ( (size == 0 && this.top == null)
      || (size > 0 && this.top != null));
}
```



## Checking Post-conditions

Assert post-conditions and invariants to inform yourself when you violate the contract.

```
   public void push(Object item)  {
      top = new Cell(item, top);
      size++;
      assert !this.isEmpty();          // post-condition
      assert this.top() == item;       // post-condition
      assert invariant();
   }
```

Check them whenever the implementation is non-trivial.



## Example

Consider int array binary search code:

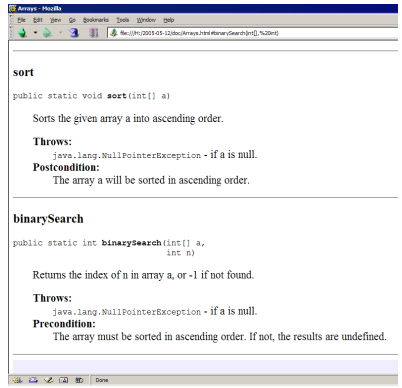
```
/** Returns index of value n in array a.
 *  @pre.condition The array a is in sorted order.
 */
public static void binarySearch(int[] a, int n) {
    assert isSorted(a) : "Array must be sorted";
    ...
}

private static boolean isSorted(int[] a) {
    for (int i = 0; i < a.length - 1; i++)
        if (a[i] > a[i+1])
            return false;
    return true;
}
```



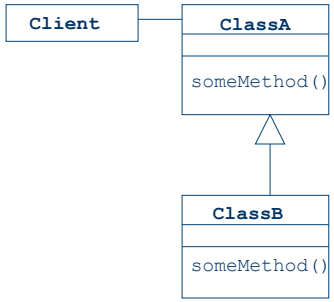
## Running Javadoc with tags

- ```
  javadoc -source 1.5
          -d output_folder_name
          -tag pre.condition:cm:"Precondition:"
          -tag post.condition:cm:"Postcondition:"
          file_name.java
  ```
- Javadoc output will show pre, postconditions


  ```

## Inheritance: Pre-conditions

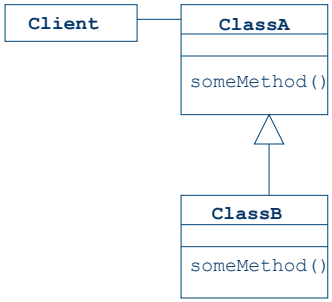
- If the precondition of the `ClassB.someMethod` is stronger than the precondition of the `ClassA.someMethod`, then this is not fair to the `Client`
- The code for `ClassB` may have been written after `Client` was written, so `Client` has no way of knowing its contractual requirements for `ClassB`






## Inheritance: Post-conditions

- 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

- Liskov Substitution Principle
  - Wherever an instance of a class is expected, an instance of one of its subclasses can be substituted.
- Therefore,
  - A subclass may keep or weaken the preconditions of an overridden method
  - A subclass may keep or strengthen the postconditions of an overridden method
  - A subclass may keep or strengthen the invariants of a subclass

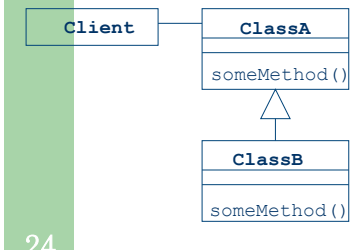


In `ClassA`:

```
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`