CS 619 Introduction to OO Design and Development

Design by Contract

Fall 2012

Why Design by Contract

What's the difference with Testing?

- Testing tries to *diagnose* (and cure) defects after the facts.
- Design by Contract tries to *prevent* certain types of defects.

"Design by Contract" falls under Implementation/Design

Design by Contract is particularly useful in an Object-Oriented context

- preventing errors in interfaces between classes
  (e.g. subclass and superclass via subcontracting)
- preventing errors while reusing classes
  (e.g. evolving systems, thus incremental and iterative development)

Use Design by Contract in combination with Testing!

What is Design By Contract ?

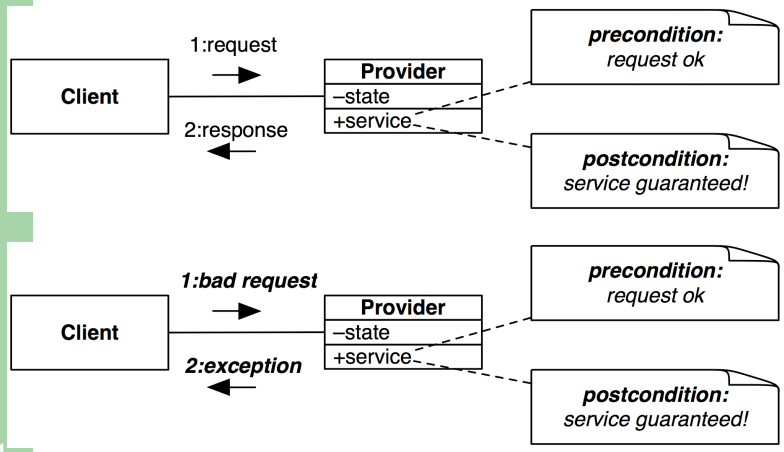
View the relationship between two classes as a formal agreement, expressing each party's rights and obligations." ([Meye97a], Introduction to Chapter 11)

- Each party expects benefits (rights) and accepts obligations
- Usually, one party's benefits are the other party's obligations
- Contract is declarative: it is described so that both parties can understand what service will be guaranteed without saying how.

Example: Airline reservation

	Obligations	Rights
Customer (Client Class)	– Be at Brussels airport at least 1 hour before scheduled departure time – Bring acceptable baggage – Pay ticket price	– Reach Chicago
Airline (Supplier Class)	– Bring customer to Chicago	– No need to carry passenger who is late, – has unacceptable baggage, – or has not paid ticket

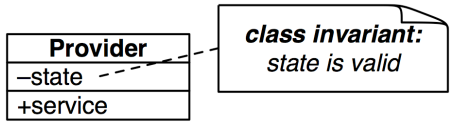
Design by Contract == *Don't accept anybody else's garbage!*



Class Invariants

Those conditions that must exist for all visible methods of a class

- Coupled to objects, not methods
- Adding an invariant is similar to adding the same constraint as a precondition and postcondition for all methods in a class



A class invariant characterizes the valid states of instances
It must hold:

1. after construction
2. before and after every public method

Method Pre- and Post-conditions

The pre-condition binds clients:

- it defines what the data abstraction requires for a call to the operation to be legitimate
- it may involve initial state and arguments
- example: stack is not empty

The post-condition, in return, binds the provider:

- it defines the conditions that the data abstraction ensures on return
- it may only involve the initial and final states, the arguments and the result
- example: size = old size + 1

- Both are coupled to individual methods

Benefits and Obligations

A contract provides benefits and obligations for both clients and providers:

	Obligations	*Benefits*
Client	Only call pop() on a non-empty stack!	Stack size decreases by 1. Top element is removed.
Provider	Decrement the size. Remove the top element.	No need to handle case when stack is empty!

Redundant Checks

Redundant checks: naive way for including contracts in the source code

```
public char pop () {
    if (isEmpty (this)) {
        ... //Error-handling
} else {
        ...}
```

This is redundant code: it is the responsibility of the client to ensure the pre-condition!

Redundant Checks Considered Harmful

- Extra complexity
  Due to extra (possibly duplicated) code
  ... which must be verified as well
- Performance penalty
  Redundant checks cost extra execution time
- Wrong context
  How severe is the fault? How to rectify the situation? A service provider cannot asses the situation, only the consumer can.

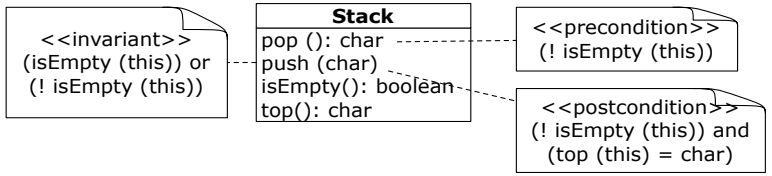
Example: Stack Specification

```
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



Programming Language Support

Eiffel

- Eiffel is designed as such ... but only used in limited cases

C++

- assert() in C++ assert.h does not throw an exception
- It's possible to mimic assertions (incl. compilation option) in C++
  - + (see "Another Mediocre Assertion Mechanism for C++" by Pedro Guerreiro in TOOLS2008)
- Documentation extraction is more difficult but feasible

Java

- ASSERT is standard since Java 1.4
  ... however limited "design by contract" only; acknowledge by Java designers
  - + see http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html
- Documentation extraction using JavaDoc annotations

Smalltalk (and other languages)

- Possible to mimic; compilation option requires language idioms
- Documentation extraction is possible (style Javadoc)

10

Assertions

- assertion = any boolean expression we expect to be true at some point.

Assertions..

- Help in writing correct software
  - ➡ formalizing invariants, and pre- and post-conditions
- Aid in maintenance of documentation
  - ➡ specifying contracts IN THE SOURCE CODE
  - ➡ tools to extract interfaces and contracts from source code
- Serve as test coverage criterion
  - ➡ Generate test cases that falsify assertions at run-time
- Should be configured at compile-time
  - ➡ to avoid performance penalties with trusted parties
  - ➡ What happens if the contract is not satisfied?

11

Assertions in Source Code

```
public char pop() throws AssertionException {
    my_assert(!this.isEmpty());
    return _store[_size--];
    my_assert(invariant());
    my_assert(true); //empty postcondition
}


private boolean invariant() {
    return (_size >= 0) && (_size <= _capacity);}


private void my_assert(boolean assertion)
    throws AssertionException {
    if (!assertion) {
        throw new AssertionException
            ("Assertion failed");}
}
```

Should be turned on/off via compilation option

12

Exception Handling

```
public class AssertionException extends Exception {
    AssertionException() { super(); }
    AssertionException(String s) { super(s); }
}


static public boolean testEmptyStack() {
    ArrayStack stack = new();
    try {
        // pop() will raise an exception
        stack.pop();


    } catch(AssertionException err) {
        // we should get here!
        return true;
    };

    return false;
}
```

If an 'AssertionException' is raised within the try block ...

... we will fall into the 'catch' block

Assertions are not...

Assertions look strikingly similar yet are different from...

- Redundant Checks
  - + Assertions become part of a class interface
  - + Compilation option to turn on/off
- Checking End User Input
  - + Assertions check software-to-software communication, not software-to-human
- Control Structure
  - + Raising an exception is a control structure mechanism
  - + ... violating an assertion is an error
    - precondition violation responsibility of the client of the class
    - postcondition violation responsibility of the supplier of the class
      - ➡ Only turn off assertions with trusted parties
      - ➡ Tests must verify whether exceptions are thrown

Assertions in Java

`assert` is a keyword in Java since version 1.4

```
assert expression;
```

will raise an `AssertionError` if `expression` is false.

Be sure to enable exceptions in eclipse! (And set the vm flag -enableassertions [-ea])

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;
}
```

When should you check pre-conditions to methods?

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();
}
```

When should you check post-conditions?
Check them whenever the implementation is non-trivial.

Preconditions, Postconditions and Class Invariants In Java Doc

```
/**
  * @invariant gpa >= 0
  */
class Student {
protected _gpa;

/**
  * GPA always >=0
  * @pre gpa >= 0
  */
void setGPA (double gpa){…}

/**
  * GPA always >=0
  * @post return >= 0
  */
double getGPA(){…}
}
```


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