

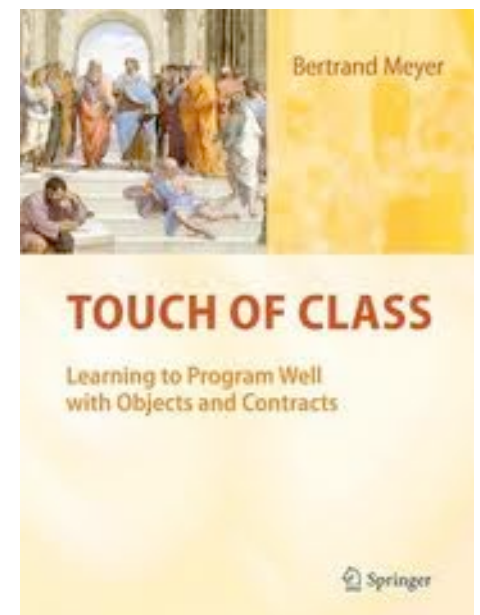
Design by Contract

Alexandre Bergel
abergel@dcc.uchile.cl
03/04/2012

Design by contract

Bertrand Meyer, *Touch of Class — Learning to Program Well with Objects and Contracts*

Springer, 2009



Design by contract

The central idea is a metaphor on how elements of a software system *collaborate* with each other, on the basis of mutual *obligations* and *benefits*

Essential for *testing*

Foundation for *exception handling*

Roadmap

- 1.Contracts, exceptions, failures, defects and assertions
- 2.Data abstraction — Stack example
- 3.Class invariants
- 4.Programming by Contract
- 5.Assertions



Roadmap

1.Contracts, exceptions, failures, defects and assertions

2.Data abstraction — Stack example

3.Class invariants

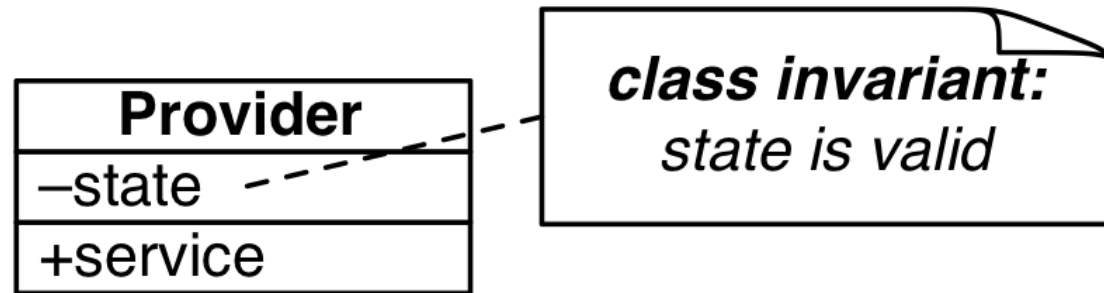
4.Programming by Contract

5.Assertions



Class Invariant

An *invariant* is a predicate that *must hold* at certain points in the execution of a program

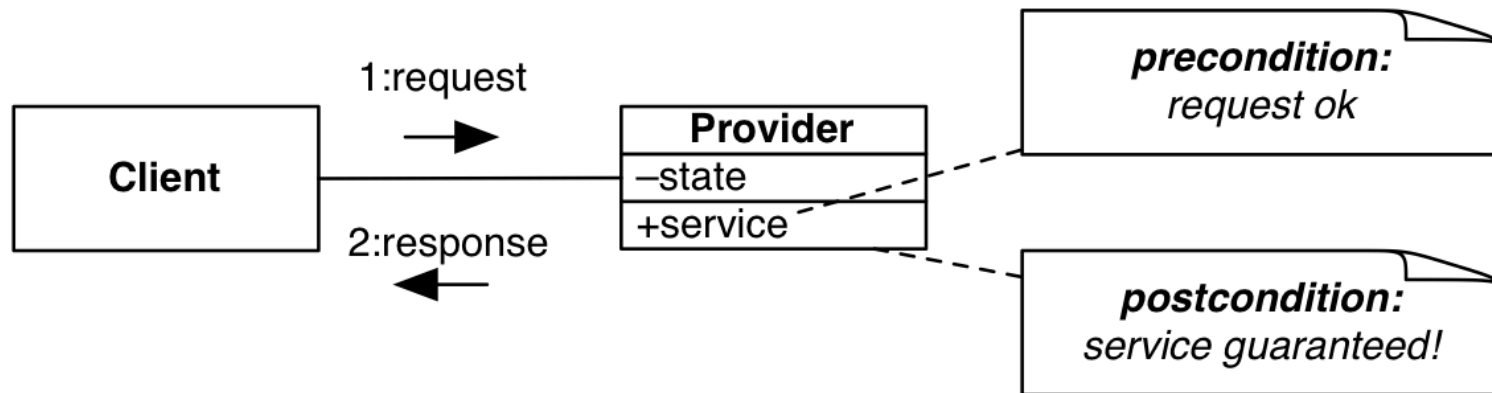


A class invariant characterizes *the valid states of instances*. It must hold

- 1 - After construction
- 2 - Before and after every public method

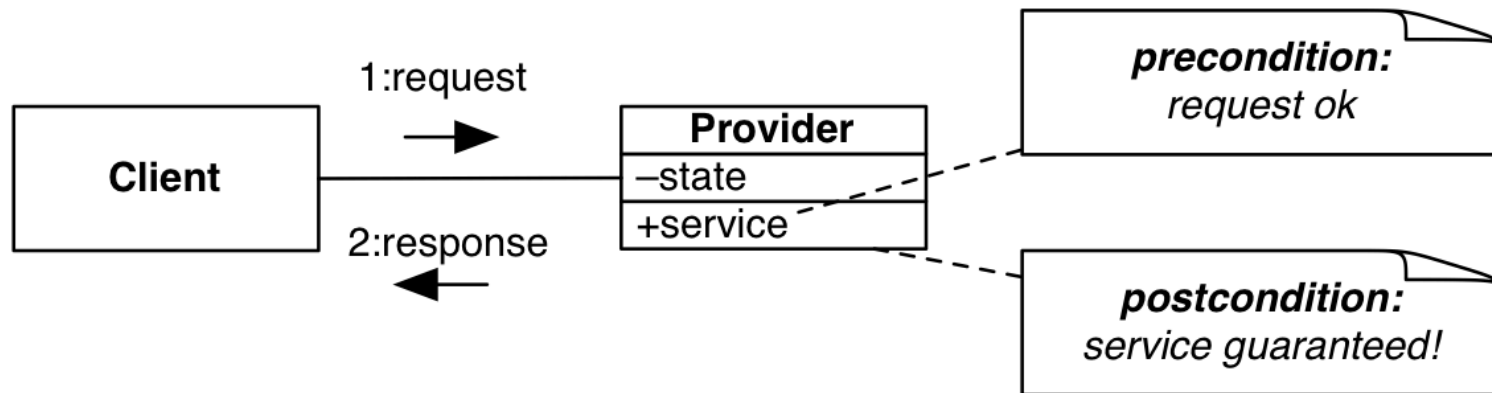
Contracts

A contract *binds the client* to pose valid requests, and *binds the provider* to correctly provide the service.



Contract violations

If either the client or the provider violates the contracts, an *exception* is raised.



NB: The service does not need to implement any special logic to handle errors. It simply raises an exception

Assertions

An assertion is a declaration of a boolean expression that the programmer believes should hold at some point in a program.

Contract are formalized/concretized with assertions

Assertions should not affect the logic of the program

If an assertion fails, an exception should be raised

```
x = y*y;  
assert x >= 0;
```

Exceptions, failures and defects

An *exception* is the occurrence of an *abnormal condition during the execution* of a software element

A *failure* is the *inability of a software element to satisfy* its purpose (e.g., assertions not evaluating at true).

A *defect* (AKA “*bug*”) is the *presence in the software* of some element *not satisfying its specification*

Contracts may fail due to defects in the client or server code. *Failure* should be signaled by raising an *exception*

Roadmap

1.Contracts, exceptions, failures, defects and assertions

2.Data abstraction — Stack example

3.Class invariants

4.Programming by Contract








5.Assertions



Stacks

A Stack is a classical data abstraction with many applications in computer programming.

Stacks support two mutating methods: push and pop

Operation	Stack	isEmpty()	size()	top()
		TRUE	0	(error)
push(6)		FALSE	1	6
push(7)		FALSE	2	7
push(3)		FALSE	3	3
pop()		FALSE	2	7
push(2)		FALSE	3	2
pop()		FALSE	2	7

Stack pre- and postconditions

Stacks should deliver the following contract:

<i>Operation</i>	<i>Requires</i>	<i>Ensures</i>
<code>isEmpty()</code>	-	<i>no state change</i>
<code>size()</code>	-	<i>no state change</i>
<code>push(Object item)</code>	-	not empty, size == old size + 1,
<code>top()</code>	not empty	<i>no state change</i>
<code>pop()</code>	not empty	size == old size

Stack invariant

The only thing we can say about the Stack class invariant is that the size is always ≥ 0

We don't know anything yet about its state

Example: Balancing Parentheses

Problem

Determine whether an expression containing parentheses `()`, brackets `[]` and braces `{ }` is correctly balanced

Examples

balanced

```
if (a.b()) { c[d].e(); }  
else { f[g][h].i(); }
```

not balanced

```
((a+b()))
```

A simple algorithm

When you read a *left* parenthesis, *push the matching parenthesis* on a stack

when you read a *right* parenthesis, *compare it* to the value on top of the stack

if they *match*, you *pop and continue*

if they *mismatch*, the expression is *not balanced*

if the *stack is empty* at the end, the whole expression is *balanced*, otherwise not

Using a Stack to match parentheses

Sample input: “([{ }]]”

<i>Input</i>	<i>Case</i>	<i>Op</i>	<i>Stack</i>
(left	push))
[left	push])]
{	left	push })]}
}	match	pop)]
]	match	pop)
]	mismatch	^false)

The ParenMatch class

A ParenMatch object *uses a stack* to check if parentheses in a text String are balanced:

```
public class ParenMatch {  
    private String line;  
    private StackInterface stack;  
  
    public ParenMatch (String aLine, StackInterface aStack)  
    {  
        line = aLine;  
        stack = aStack;  
    }  
}
```

A declarative algorithm

We implement our algorithm at a high level of abstraction:

```
public boolean parenMatch() {
    for (int i=0; i<line.length(); i++) {
        char c = line.charAt(i);
        if (isLeftParen(c)) { // expect matching right paren later
            stack.push(matchingRightParen(c)); // Autoboxed to Character
        } else {
            if (isRightParen(c)) {
                // empty stack => missing left paren
                if (stack.isEmpty()) { return false; }
                if (stack.top().equals(c)) { // Autoboxed
                    stack.pop();
                } else { return false; } // mismatched paren
            }
        }
    }
    return stack.isEmpty(); // not empty => missing right paren
}
```

Ugly, procedural version

```
public boolean parenMatch() {
    char[] chars = new char[1000]; // ugly magic number
    int pos = 0;
    for (int i=0; i<line.length(); i++) {
        char c = line.charAt(i);
        switch (c) { // what is going on here?
            case '{' : chars[pos++] = '}'; break;
            case '(' : chars[pos++] = ')'; break;
            case '[' : chars[pos++] = ']'; break;
            case ']' : case ')' : case '}' :
                if (pos == 0) { return false; }
                if (chars[pos-1] == c) { pos--; }
                else { return false; }
                break;
            default : break;
        }
    }
    return pos == 0; // what is this?
}
```

Helper methods

The helper methods are trivial to implement, and their details only get in the way of the main algorithm.

```
private boolean isLeftParen(char c) {  
    return (c == '(') || (c == '[') || (c == '{');  
}  
  
private boolean isRightParen(char c) {  
    return (c == ')') || (c == ']') || (c == '}');  
}
```

What is Data Abstraction?

An implementation of a stack consists of:

- a *data structure* to represent the state of the stack

- a *set of operations* that access and modify the stack

Encapsulation means *bundling together related entities*

Information hiding means *exposing an abstract interface* and hiding the rest

An Abstract Data Type (ADT):

- encapsulates data and operations, and

- hides the implementation behind a well-defined interface

Abstract Data Type

Note that there is not only one definition of ADT

What we will see is slightly different than Cook's formulation

William R. Cook, "On Understanding Data Abstraction, Revisited",
OOPSLA'09

StackInterface

Interfaces let us *abstract* from concrete implementations:

```
public interface StackInterface {  
    public boolean isEmpty();  
    public int size();  
    public void push(Object item);  
    public Object top() ;  
    public void pop();  
}
```

How can clients accept multiple implementations of an ADT?

Make them depend only on an interface or an abstract class.

Interfaces in Java

Interfaces reduce coupling between objects and their clients:

A class can implement *multiple interfaces* ... but can only *extend one parent class*

Clients should *depend on an interface*, not an implementation ... so implementations don't need to extend a specific class

Define an interface for any ADT that will have more than one implementation

Why are ADTs important?

Communication — Declarative Programming

An ADT exports what a client needs to know, and nothing more!

By using ADTs, you communicate what you want to do, *not how to do it!*

ADTs allow you to directly *model your problem domain* rather than how you will use to the computer to do so

Why are ADTs important?

Software Quality and Evolution

ADTs help you to *decompose a system into manageable parts*, each of which can be separately implemented and validated.

ADTs *protect clients from changes* in implementation.

ADTs encapsulate client/server *contracts*

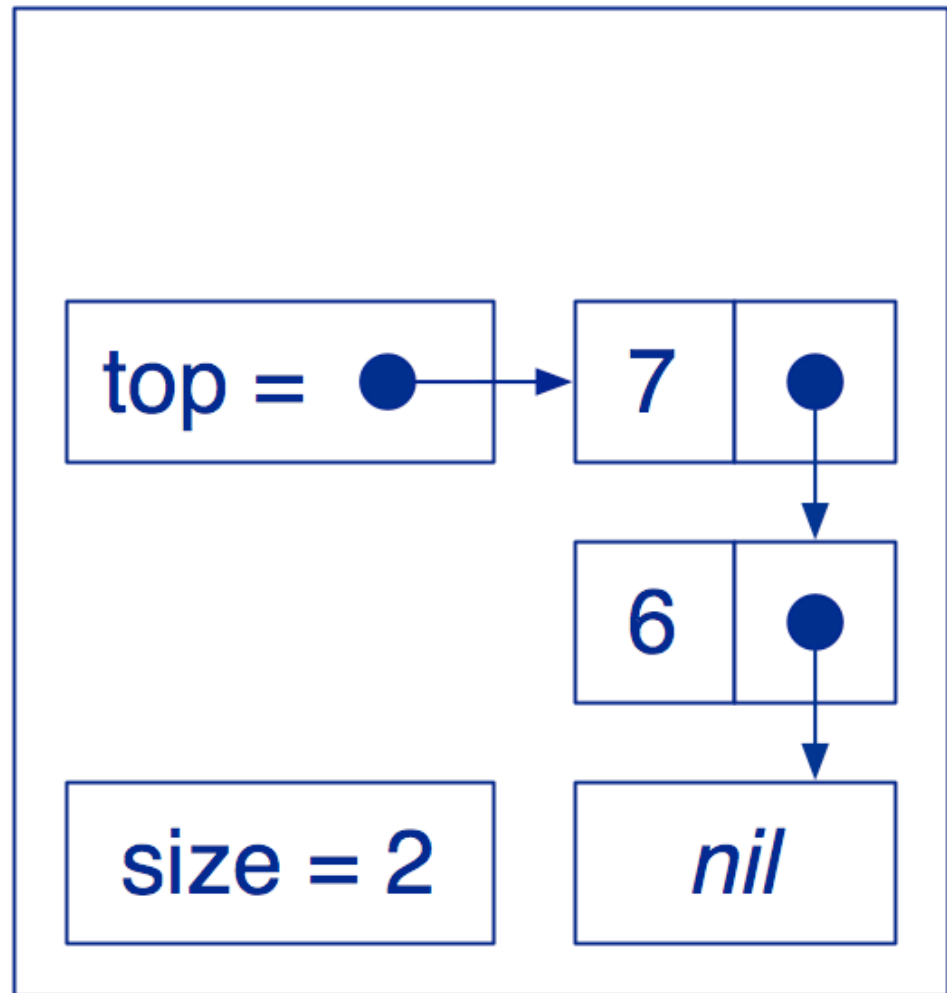
Interfaces to ADTs can be *extended* without affecting clients.

New implementations of ADTs can be *transparently added* to a system

Stacks as Linked Lists

A Stack can easily be implemented by a linked data structure:

```
stack = new Stack();  
stack.push(6);  
stack.push(7);  
stack.push(3);  
stack.pop();
```



LinkStack Cells

We can define the Cells of the linked list as an inner class within LinkStack:

```
public class LinkStack implements StackInterface {  
    private Cell top;  
    private class Cell {  
        Object item;  
        Cell next;  
        Cell(Object item, Cell next) {  
            this.item = item;  
            this.next = next;  
        }  
    }  
    ...  
}
```

Private vs Public instance variables

When should instance variables be public?

Always make instance variables private or protected.

The Cell class is a special case, since its instances are strictly private to LinkStack!

Roadmap

1.Contracts, exceptions, failures, defects and assertions

2.Data abstraction — Stack example

3.Class invariants

4.Programming by Contract

5.Assertions



LinkStack ADT

The constructor must construct a *valid initial state*:

```
public class LinkStack implements StackInterface {  
    ...  
    private int size;  
    public LinkStack() {  
        // Establishes the class invariant.  
        top = null;  
        size = 0;  
    }  
    ...  
}
```


Class Invariants

A class invariant is any condition that expresses the *valid states* for objects of that class:

it must be *established* by every constructor

every public method

may *assume* it holds when the method starts

must *re-establish* it when it finishes

Stack instances must satisfy the following invariant:

size ≥ 0

...

LinkStack Class Invariant

A valid LinkStack instance has an integer size, and a top that points to a sequence of linked Cells, such that:

size is always ≥ 0

When size is zero, top points nowhere (`== null`)

When size > 0 , top points to a Cell containing the top item

When to check invariants?

In principle, check invariants:

at the end of each *constructor*

at the end of every *public method*

Where to check invariants?

Invariants can be checked:

on the client site: the class embeds a method called `invariant()` that the programmer should invoke. The invariant is then always checked.

in unit tests: invariant is then checked only for a set of particular scenarios. This is what we will see at the next lecture.

Roadmap

1.Contracts, exceptions, failures, defects and assertions

2.Data abstraction — Stack example

3.Class invariants

4.Programming by Contract

5.Assertions



Design by Contract

Every ADT is designed to provide certain *services* given certain *assumptions* hold

An ADT establishes a contract with its clients by associating a precondition and a postcondition to every operation O , which states:

if the precondition does not hold, the ADT is not required to provide anything!

In other words...

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

Pre- and Postconditions

The precondition binds clients:

it defines what the ADT requires for a call to the operation to be legitimate

it may involve initial state and arguments

example: stack is not empty

The postcondition, in return, binds the supplier:

it defines the conditions that the ADT ensures on return

it may only involve the initial and final states, the arguments and the result

example: $\text{size} = \text{old size} + 1$

Benefits and Obligations

A contract provides *benefits and obligations* for both clients and suppliers:

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

Roadmap

1.Contracts, exceptions, failures, defects and assertions

2.Data abstraction — Stack example

3.Class invariants

4.Programming by Contract

5.Assertions



Assertions

An assertion is any boolean expression we expect to be true at some point

assert is a keyword in Java

```
assert expression;
```

will raise an AssertionError if expression is false.

NB: Throwable Exceptions must be declared; Errors and runtime errors (including AssertionError) do not need to be!

Assertions have four principle applications

Help in writing correct software

formalizing invariants, and pre- and post-conditions

Documentation aid

specifying contracts

Debugging tool

testing assertions at run-time

Support for software fault tolerance

detecting and handling failures at run-time

Testing Invariants

Every class has its own invariant:

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

Why protected and not private?

Disciplined Exceptions

There are only two reasonable ways to react to an exception:

- clean up the environment and *report failure* to the client
("organized panic")

- attempt to change the conditions that led to failure and *retry*

It is not acceptable to return control to the client without special notification

Disciplined Exceptions

When should an object throw an exception?

If and only if an assertion is violated

If it is not possible to run your program without raising an exception, then you are abusing the exception-handling mechanism!

Special case for graphical user interfaces and stream manipulation

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 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.

Running parenMatch

```
public static void parenTestLoop(StackInterface stack) {
    BufferedReader in =
        new BufferedReader(new InputStreamReader(System.in));
    String line;
    try {
        System.out.println("Please enter parenthesized expressions to test");
        System.out.println("(empty line to stop)");
        do {
            line = in.readLine();
            System.out.println(new ParenMatch(line, stack).reportMatch());
        } while(line != null && line.length() > 0);
        System.out.println("bye!");
    } catch (IOException err) {
    } catch (AssertionException err) {
        err.printStackTrace();
    }
}
```

Running ParenMatch.main ...

```
Please enter parenthesized expressions to test
(empty line to stop)
(hello) (world)
"(hello) (world)" is balanced
()
"()" is balanced
static public void main(String args[]) {
"static public void main(String args[]) {" is not balanced
()
"()" is not balanced
}
"}" is balanced

"" is balanced
bye!
```

Which contract is
being violated?

What you should know!

What is the difference between *encapsulation* and *information* hiding?

What is an *assertion*?

How are contracts *formalized* by pre- and post-conditions?

What is a class invariant and how can it be specified?

What are assertions useful for?

How can *exceptions* be used to improve program robustness?

What situations may cause an *exception* to be *raised*?

Can you answer these questions?

Why is *strong coupling* between clients and suppliers a bad thing?

When should you *call super()* in a constructor?

When should you use an inner class?

How would you write a general `assert()` method that works for any class?

What happens when you `pop()` an empty `java.util.Stack`? Is this good or bad?

What *impact* do assertions have on *performance*?

Can you implement the missing `LinkStack` methods?

License

<http://creativecommons.org/licenses/by-sa/2.5>



Attribution-ShareAlike 2.5

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:



Attribution. You must attribute the work in the manner specified by the author or licensor.



Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.