3. Design by Contract

Oscar Nierstrasz
Design by Contract

Roadmap

- Contracts
- Stacks
- Design by Contract
- A Stack Abstraction
- Assertions
- Example: balancing parentheses
Roadmap

> Contracts
> Stacks
> Design by Contract
> A Stack Abstraction
> Assertions
> Example: balancing parentheses
Class Invariants

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
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 contract, an *exception* is raised.

**NB:** The service does not need to implement any special logic to handle errors — it simply raises an exception!
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.

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

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

1. clean up the environment and report failure to the client (“organized panic”)
2. attempt to change the conditions that led to failure and retry

A failed assertion often indicates presence of a software defect, so “organized panic” is usually the best policy.
Roadmap

> Contracts
> **Stacks**
> Design by Contract
> A Stack Abstraction
> Assertions
> Example: balancing parentheses
A Stack is a classical data abstraction with many applications in computer programming. **Stacks support two mutating methods: push and pop.**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack</th>
<th>isEmpty()</th>
<th>size()</th>
<th>top()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TRUE</td>
<td>0</td>
<td>(error)</td>
</tr>
<tr>
<td>push(6)</td>
<td>[6]</td>
<td>FALSE</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>push(7)</td>
<td>[6, 7]</td>
<td>FALSE</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>push(3)</td>
<td>[6, 7, 3]</td>
<td>FALSE</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>pop()</td>
<td>[6, 7]</td>
<td>FALSE</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>push(2)</td>
<td>[6, 7, 2]</td>
<td>FALSE</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>pop()</td>
<td>[6, 7]</td>
<td>FALSE</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>
# Stack pre- and postconditions

Stacks should respect the following contract:

<table>
<thead>
<tr>
<th>Service</th>
<th>Pre</th>
<th>Post</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isEmpty()</code></td>
<td>-</td>
<td>no state change</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>-</td>
<td>no state change</td>
</tr>
<tr>
<td><code>push(Object item)</code></td>
<td>not empty</td>
<td>not empty, size == old size + 1, top == item</td>
</tr>
<tr>
<td><code>top()</code></td>
<td>not empty</td>
<td>no state change</td>
</tr>
<tr>
<td><code>pop()</code></td>
<td>not empty</td>
<td>size == old size - 1</td>
</tr>
</tbody>
</table>
Stack invariant

> The only thing we can say about the Stack class invariant is that the size is always $\geq 0$ —we don’t know anything yet about its state!
Roadmap

- Contracts
- Stacks
- Design by Contract
- A Stack Abstraction
- Assertions
- Example: balancing parentheses
Design by Contract

When you design a class, each service \( S \) provided must specify a clear contract.

“If you promise to call \( S \) with the precondition satisfied, then I, in return, promise to deliver a final state in which the post-condition is satisfied.”

Consequence:
— if the precondition does not hold, the object is not required to provide anything! (in practice, an exception is raised)
In other words …

Design by Contract =

*Don’t accept anybody else’s garbage!*
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
A contract provides *benefits and obligations* for both clients and providers:

<table>
<thead>
<tr>
<th><strong>Obligations</strong></th>
<th><strong>Benefits</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td></td>
</tr>
<tr>
<td>Only call pop() on a non-empty stack!</td>
<td>Stack size decreases by 1. Top element is removed.</td>
</tr>
<tr>
<td><strong>Provider</strong></td>
<td></td>
</tr>
<tr>
<td>Decrement the size. Remove the top element.</td>
<td>No need to handle case when stack is empty!</td>
</tr>
</tbody>
</table>
Roadmap

- Contracts
- Stacks
- Design by Contract
- **A Stack Abstraction**
- Assertions
- Example: balancing parentheses
Interfaces let us *abstract* from concrete implementations:

```java
public interface StackInterface<E> {
    public boolean isEmpty();
    public int size();
    public void push(E item);
    public E top();
    public void pop();
}
```

💡 **How can clients accept multiple implementations of a data abstraction?**

✔ *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 data abstraction that will have more than one implementation*
Stacks as Linked Lists

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

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

![Diagram of a stack implementation as a linked list]

*top = 7, size = 2, nil*
We can define the Cells of the linked list as an *inner class* within LinkStack:

```java
public class LinkStack<E> implements StackInterface<E> {
    private Cell top;
    private class Cell {
        E item;
        Cell next;
        Cell(E item, Cell next) {
            this.item = item;
            this.next = next;
        }
        ...
    }
    ...
}
```
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!
The constructor must construct a *valid initial state*:

```java
public class LinkStack<E> implements StackInterface<E> {
    ...
    private int size;
    public LinkStack() {
        // Establishes the class invariant.
        top = null;
        size = 0;
    }
    ...
```
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 $\geq 0$
- ...
A valid LinkStack instance has an integer size, and a top that points to a sequence of linked cells, such that:

— size is always $\geq 0$
— 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 mutator*
Roadmap

- Contracts
- Stacks
- Design by Contract
- A Stack Abstraction
- **Assertions**
- Example: balancing parentheses
> An assertion is a declaration of a boolean expression that the programmer believes must hold at some point in a program.

— Assertions should not affect the logic of the program
— If an assertion fails, an exception is raised

```java
x = y*y;
assert x >= 0;
```
Assertions have four principle applications:

1. Help in writing correct software
   — formalizing invariants, and pre- and post-conditions
2. Documentation aid
   — specifying contracts
3. Debugging tool
   — testing assertions at run-time
4. Support for software fault tolerance
   — detecting and handling failures at run-time
assert is a keyword in Java since version 1.4

```java
assert expression;
```

will raise an AssertionError if expression is false.

— NB: Throwable Exceptions must be declared; Errors need not be!

✔ Be sure to enable exceptions in eclipse! (And set the vm flag -enableassertions [-ea])
Enabling assertions in eclipse
Checking pre-conditions

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

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

*NB: This is all you have to do!*

✎ 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:

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

Why protected and not private?
Checking post-conditions

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

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

> **NB:** This is all you have to do!

**When should you check post-conditions?**

✔ *Check them whenever the implementation is non-trivial.*
Roadmap

- Contracts
- Stacks
- Design by Contract
- A Stack Abstraction
- Assertions
- **Example: balancing parentheses**
Example: Balancing Parentheses

**Problem:**
> Determine whether an expression containing parentheses ( ), brackets [ ] and braces { } is correctly balanced.

**Examples:**
> balanced:

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

> not balanced:

```plaintext
((a+b()))
```
A simple algorithm

**Approach:**

> 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: “( [ { } ] ]”

<table>
<thead>
<tr>
<th>Input</th>
<th>Case</th>
<th>Op</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>left</td>
<td>push</td>
<td>)</td>
</tr>
<tr>
<td>[</td>
<td>left</td>
<td>push</td>
<td>]]</td>
</tr>
<tr>
<td>{</td>
<td>left</td>
<td>push</td>
<td>]]}}</td>
</tr>
<tr>
<td>}</td>
<td>match</td>
<td>pop</td>
<td>]]</td>
</tr>
<tr>
<td>]</td>
<td>match</td>
<td>pop</td>
<td>)</td>
</tr>
<tr>
<td>]</td>
<td>mismatch</td>
<td>^false</td>
<td>)</td>
</tr>
</tbody>
</table>
The ParenMatch class

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

```java
public class ParenMatch {
    private String line;
    private StackInterface<Character> stack;

    public ParenMatch(String aLine,
                       StackInterface<Character> aStack) {
        line = aLine;
        stack = aStack;
    }
```
A declarative algorithm

We implement our algorithm at a high level of abstraction:

```java
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.peek().equals(c)) { // Autoboxed
                    stack.pop();
                } else { return false; } // mismatched paren
            }
        }
    }
    return stack.isEmpty(); // not empty => missing right paren
}
```
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.

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

private boolean isRightParen(char c) {
    return (c == ')') || (c == ']') || (c == '}');
}
```
public static void parenTestLoop(StackInterface<Character> 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 has been violated?
What you should know!

✎ What is an abstract data type?
✎ What is the difference between encapsulation and information hiding?
✎ 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?
✎ What situations may cause an exception to be raised?
✎ How can helper methods make an implementation more declarative?
Can you answer these questions?

- When should you call super() in a constructor?
- When should you use an inner 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?
Attribution-ShareAlike 3.0

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.

[http://creativecommons.org/licenses/by-sa/3.0/]