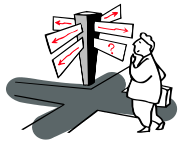
CHAPTER 6 – Design by Contract



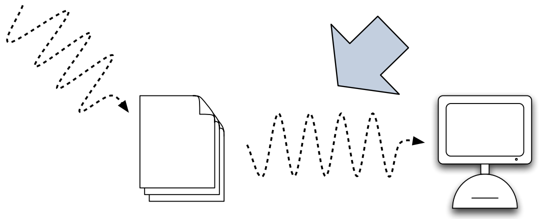
Literature

- [Ghez02a], [Somm04a], [Pres00a]
  + Occurences of “contract”, “assertions”, “pre” and “postconditions”, via index
- [Meye97a] Object-Oriented Software Construction, B. Meyer, Prentice Hall, Second Edn., 1997.
  + An excellent treatment on the do’s and don’ts of object-oriented development. Especially relevant are the chapters 6, 11-12

Copies of the following two articles are available from the course web-site.

- [Jeze97a] “Put it in the contract: The lessons of Ariane”, Jean-Marc Jézéquel and Bertrand Meyer, IEEE Computer, January 1997, Vol30, no. 2, pages 129-130. A slightly different version of this article is available at http://www.irisa.fr/pampa/EPEE/Ariane5.html
  + A (heatedly debated) column arguing that Design by Contract would have prevented the crash of the first Ariane5 missile.
- [GarlXXa] “Critique of ‘Put it in the contract: The lessons of Ariane’”, Ken Garlington. See http://home.flash.net/~kennieg/ariane.html
  + An article providing counterarguments to the former. Yet by doing so gives an excellent comparison with Testing and Code Inspection.

When Design by Contract ?



Mistakes are possible (likely !?)

- while transforming requirements into a system
- while system is changed during maintenance

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 (incl. subclass and superclass via subcontracting)
- preventing errors while reusing classes (incl. evolving systems, thus incremental and iterative development)
  ➡ Example of the Ariane 5 crash

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

Pre- and Post-conditions + Invariants

obligations are expressed via pre- and post-conditions

"If you promise to call me with the precondition satisfied, then I, in return promise to deliver a final state in which the postcondition is satisfied."

pre-condition: {x >= 9}

component: {x := x + 5}

post-condition: {x >= 13}

... and invariants

"For all calls you make to me, I will make sure the invariant remains satisfied."

invariant: {x >= y}

- pre-condition: {x > 0, y > 0}
- component: {x := x + y}

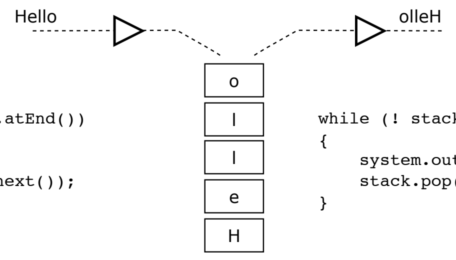
- pre-condition: {x > 0, y < 0}

component: {x := x - y}

Quiz: Whose fault is it when a pre-condition is NOT satisfied ?

Example: Stack

Given
A stream of characters, length unknown
Requested
Produce a stream containing the same characters but in reverse order
Specify the necessary intermediate abstract data structure



```
while (! inStream.atEnd())
{
    stack.push (
        inStream.next());
}
```

```
while (! stack.isEmpty())
{
    system.out.print (
    stack.pop());
}
```

Example: Stack Specification

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

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

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

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

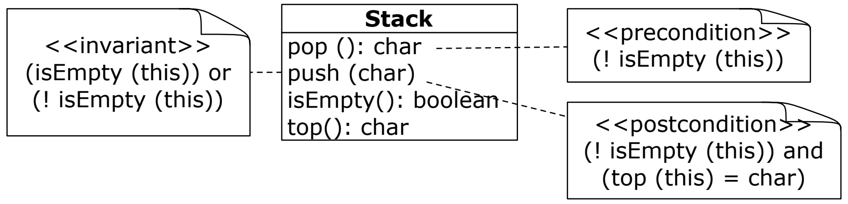
```
    public void push (char)
        require: true
        ensure: (! isEmpty (this))
            and (top (this) = char)
```

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

```
    public void top (char) : char
        require: ...
        ensure: ...
    public void isEmpty () : boolean
        require: ...
        ensure: ...
```

Left as an exercise

Design by Contract in UML



So what: isn't this pure documentation?
Who will
(a) Register these contracts for later reference (the book of laws)?
(b) Verify that the parties satisfy their contracts (the police)?
Answer
(a) The source code
(b) The running system
Quiz: What happens when a pre-condition is NOT satisfied ?

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.

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?

Quiz: What happens when a pre-condition is NOT satisfied ?
- = What should an object do if an assertion does not hold?
  ➡ Throw an exception.

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

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

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)

Two Implementation Issues

1) The use of 'previous' or 'old' state
- sometimes postconditions compare exit state with starting state

```
public char pop ()
  require: ! isEmpty (this)
  ensure: (top (old) = char)
    and (size (old) = size (this) + 1)
```

Use 'old' as a way to refer to the starting state of the receiver

- Eiffel has a pseudo variable 'old'
- Mimicking assertions in other languages ?
  - + store 'old' state in temporary variables

2) Invoking operations within assertions
- Assertions may invoke operations with pre- and postconditions
  - + overhead + cycles lead to infinite loops
- Eiffel switches off assertions when checking assertions
- Mimicking assertions in other languages ?
  - + Cumbersome using language idioms and class variable
    … best to avoid cycles

Testing Issues

- Pre- and post-conditions are part of the interface of a component.
  - + Part of black-box testing, not white-box testing
    - - Do *not* include assertions in basis-path testing
    - - Borderline case: include assertions in condition testing

- Example

```
public char pop() throws AssertionException {
    assert(!this.isEmpty());
    return _store[_size--];
}
```

- basis-path testing: cyclomatic complexity = 1; 1 path can cover the control-flow
  - + (test case 1 = non-empty stack / value on the top)
- condition testing: 2 inputs cover all conditions
  - + (test case 1 = non-empty stack / value on the top
  - + (test case 2 = empty stack / assertion exception)

Compiler Checks ?

How much assertion monitoring is needed?

All
Especially during development
Too costly during production runs

None
Fully trusted system
Metaphor “sailing without life-jacket”

Rule of thumb

- *** At least monitor the pre-conditions.
  - + Make sure that verifying pre-conditions is fast!
  - + Do not rely on switching off monitoring to gain efficiency
  - + Profile performance to see where you loose efficiency
    - - First do it, then do it right, then do it fast!

Correctness Formula

a.k.a. *Hoare triples*

Let:
A be an *operation* (defined on a class C)
{P} and {Q} are *properties* (expressed via predicates, i.e functions returning a boolean)

Then:
{P} A {Q}
is a *Correctness Formula* meaning
“Any execution of A starting in a state where P holds, will terminate in a state where Q holds”

Example: x is Integer
{x >= 9} x := x + 5 {x >=13}

Weak and Strong

(Note: “weaker” and “stronger” follow from logic theory)

Let {P1} and {P2} be conditions expressed via predicates

- {P1} is stronger then {P2} iff
  - + {P1} <> {P2}
  - + {P1} => {P2}
- example
  - + {x >= 9} is stronger then {x >= 3}

- {false} is the strongest possible condition
  [(not {false}) or {X} is always true]
- {true} is the weakest possible condition
  [(not {X}) or {true} is always true]

Remember: {P1} => {P2} is the same as (not {P1}) or {P2}

P1	TRUE	FALSE	TRUE	FALSE
P2	TRUE	TRUE	FALSE	FALSE
{P1} => {P2}	TRUE	TRUE	FALSE	TRUE
(not {P1}) or {P2}	TRUE	TRUE	FALSE	TRUE

Weak and Strong: Quiz

{P} A {Q} is a specification for operation A
- You, as a developer of A must guarantee that once {P} is satisfied and A is invoked it will terminate in a situation where {Q} holds
- If you are a lazy developer, would you prefer
  - + a weak or a strong precondition {P}?
  - + a weak or a strong postcondition {Q}?

COUNT VOTES	weak	strong	don't know
precondition {P}			
psotcondition {Q}			

Weak or Strong (Preconditions)

Given correctness formula: {P} A {Q}

If you are a lazy developer, would you prefer a weak or a strong precondition {P}?
- weak {P} => the starting situation is not constrained
- strong {P} => little cases to handle inside the operation
  - ➡ The stronger the precondition, the easier it is to satisfy the postcondition

Easiest Case
- {false} A {...}
  {false} => {X} is true for any X
  [because (not {false}) or {X} is always true]
  - + if {...} does not hold after executing A, you can blame somebody else because the precondition was not satisfied
  - + ... independent of what happens inside A

Quiz: If you are client of that class, would you prefer a weak or strong precondition ?

Weak or Strong (Postconditions)

Given correctness formula: {P} A {Q}

If you are a lazy developer, would you prefer a weak or a strong postcondition {Q}?
- weak {Q} => the final situation is not constrained
- strong {Q} => you have to deliver a lot when implementing A
  - ➡ The weaker the postcondition, the easier it is to satisfy that postcondition

Easiest Case
- {...} A {true}
  {X} => {true} is true for any X
  [because (not {X}) or {true} is always true]
  - + {true} will always hold after executing A
  - + ... given that A *terminates* in a finite time

Quiz: If you are client of that class, would you prefer a weak or strong postcondition ?

Weak or Strong (Pre- vs. Post-conditions)

Remember
- {false} A {...} is easier to satisfy then {...} A {true}
  - + With the strong precondition you may go in an infinite loop
  - + The weak postcondition must be satisfied in finite time

Invariants

Invariants correspond to the general clauses in a legal contract, i.e. properties that always must be true for a given domain.

{I} is an invariant for class C

- After invoking a constructor of C, {I} is satisfied
  - + Default constructors as well!
- All public operations on C guarantee {I} when their preconditions are satisfied

Thus, for each operation A defined on class C with invariant {I}

- {P} A {Q} should be read as{I and P} A {I and Q}
  - + strengthens the precondition => implementing A becomes easier
  - + strengthens the postcondition => implementing A becomes more difficult

Contracts and Inheritance

```
class C with invariant {I}
   and operations {Pi} mi {Qi} where i: 1 .. n
class C' extends C with invariant {I'}
   and operations {Pi'} mi {Qi'} where i: 1 .. n
```

[We ignore the case where C' extends the interface of C]

Quiz: What's the relationship between the contract in C and the contract in C'

- Invariant: Is {I'} stronger, weaker or equal to {I}
- Precondition: Is {P'} stronger, weaker or equal to {P}
- Postcondition: Is {Q'} stronger, weaker or equal to {Q}

Answer according to the *Liskov Substitution Principle*

- *** You may substitute an instance of a subclass for any of its superclasses.

VOTES	stronger	weaker	equal	don't know
{I'} vs. {I}				
{P'} vs. {P}				
{Q'} vs. {Q}				

Sidetrack: ACM Turing Award Barbara Liskov

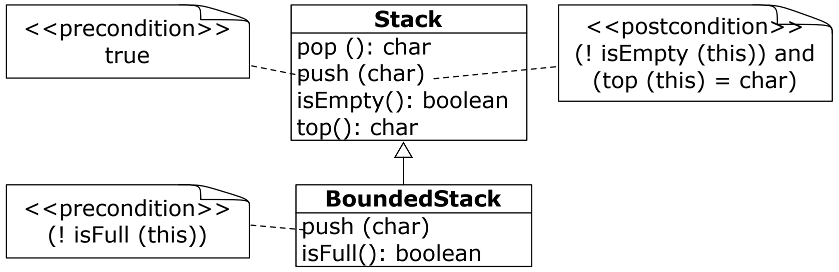
Press release — NEW YORK, March 10, 2009
– ACM, the Association for Computing Machinery

The ACM has named Barbara Liskov of the Massachusetts Institute of Technology (MIT) the winner of the 2008 ACM A.M. Turing Award. The award cites Liskov for her foundational innovations to designing and building the pervasive computer system designs that power daily life. Her achievements in programming language design have made software more reliable and easier to maintain. They are now the basis of every important programming language since 1975, including Ada, C++, Java, and C#. The Turing Award, widely considered the "Nobel Prize in Computing," is named for the British mathematician Alan M. Turing. The award carries a $250,000 prize, with financial support provided by Intel Corporation and Google Inc.

[…]

In another exceptional contribution, Liskov designed the CLU programming language, an object-oriented language incorporating "clusters" to provide coherent, systematic handling of abstract data types, which are comprised of a set of data and the set of operations that can be performed on the data. She and her colleagues at MIT subsequently developed efficient CLU compiler implementations on several different machines, an important step in demonstrating the practicality of her ideas. Data abstraction is now a generally accepted fundamental method of software engineering that focuses on data rather than processes, often identified as "modular" or "object-oriented" programming.

Contracts and Inheritance: Example (1/2)



- A client of Stack assumes a "true" pre-condition on push()
  - + Any invocation on push() will deliver the post-condition
- However, subsituting a BoundedStack adds pre-condition "! isFull(this)"
  - + BoundedStack requires more from its clients
  - + You cannot substitute a BoundedStack for a Stack

Contracts and Inheritance: Example (2/2)

As an illustration of the substitution principle, assume the following (test)code

testStack should work for any s: stack we pass !

```
testStack (s: Stack) {
    push(s, 99);
    if empty(s) {
        error(postC)};
    if pop(s) <> 99 {
        error(postC)};
}
```

```
BoundedStack s;
s= new BoundedStack(3);
push(s, 1);
push(s, 2);
push(s, 3);
testStack (s);
```

However, it runs into a pre-condition error when we pass a bounded stack that is almost full.

⇒ substitution principle is *not* satisfied

Subclassing and Subcontracting

Rule
- A subclass is a subcontractor of its parent class: it must at least satisfy the same contract
  or
- If you subcontract, you must be willing to do the job under the original conditions, no less

Thus
- Invariant: {I'} = {I}
  Invariant must remain equal (though may be expressed differently)
- Precondition: {P'} is weaker or equal to {P}
- Postcondition: {Q'} is stronger or equal to {Q}

Implementation Issue
- Eiffel has special syntax for extensions of pre- and postconditions
  ➡ Compile-time guarantee that the substitution principle holds
- In other languages it is left to the programmer to ensure this rule

How Detailed Should the Contract Be?

Given correctness formula: {P} A {Q} for operation A
- P := {false} is not desirable; nobody will invoke an operation like that
- P := {true} looks promising... at first sight
  + A will do some computation + check for abnormal cases + take corrective actions and notify clients + produce a result anyway
    ➡ It will be difficult to implement A correctly
    ➡ It will be difficult to reuse A

***** Strong preconditions make a component more reusable**

Reasonable precondition: When designing a component with preconditions
- It must be possible to justify the need for the precondition in terms of the requirements specification only
- Clients should be able to satisfy and check the precondition
  + All operations used inside the precondition should be declared public

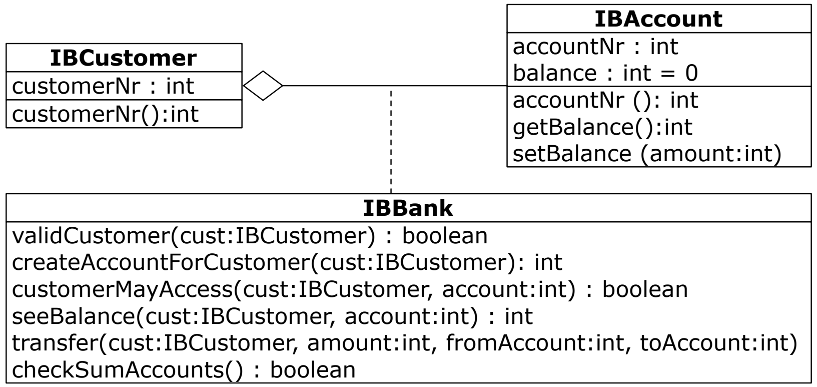
[Answer to: If you are client of that class, would you prefer a weak precondition ?]

Example: Banking - Requirements

- a bank has customers
- customers own account(s) within a bank
- with the accounts they own, customers may
  + deposit / withdraw money
  + transfer money
  + see the balance

Non-functional requirements
- *secure*: only authorised users may access an account
- *reliable*: all transactions must maintain consistent state

Example: Banking - Class Diagram


IBAccount
accountNr : int
balance : int = 0
accountNr (): int
getBalance():int
setBalance (amount:int)
IBCustomer
customerNr : int
customerNr():int
IBBank
validCustomer(cust:IBCustomer) : boolean
createAccountForCustomer(cust:IBCustomer): int
customerMayAccess(cust:IBCustomer, account:int) : boolean
seeBalance(cust:IBCustomer, account:int) : int
transfer(cust:IBCustomer, amount:int, fromAccount:int, toAccount:int)
checkSumAccounts() : boolean

Example: Banking - Contracts

Ensure the "secure" and "reliable" requirements.

```
IBBank
   invariant: checkSumAccounts()


IBBank::createAccountForCustomer(cust:IBCustomer): int
   precondition: validCustomer(cust)
   postcondition: customerMayAccess(cust, <<result>>)


IBBank::seeBalance(cust:IBCustomer, account:int) : int
   precondition: (validCustomer(cust)) AND
      (customerMayAccess(cust, account))
   postcondition: true


IBBank::transfer(cust:IBCustomer, amount:int, fromAccount:int, toAccount:int)
   precondition: (validCustomer(cust))
      AND (customerMayAccess(cust, fromAccount))
      AND (customerMayAccess(cust, toAccount))
   postcondition: true
```

Example: Banking - CheckSum

Bookkeeping systems always maintain two extra accounts, "incoming" and "outgoing"
• ⇒ the sum of the amounts of all transactions is always 0 ⇒ consistency check

Incoming	
date	amount
1/1/2000	-100
1/2/2000	-200

MyAccount	
date	amount
1/1/2000	+100
1/2/2000	+200
1/3/2000	-250

OutGoing	
date	amount
1/3/2000	+250

Correctness & Traceability

Design by contract prevents defects
Testing detect defects

➡ One of them should be sufficient!?

Design by contract and testing are *complementary*

- None of the two guarantee correctness ...
  but the sum is more than the parts.
    + Testing detects wide range of coding mistakes
    + ... design by contract prevents specific mistakes
      (due to incorrect assumptions between provider and client)
- design by contract => black box testing techniques
    + especially, equivalence partitioning & boundary value analysis
- (condition) testing => verify whether parties satisfy their obligations
    + especially, whether all assertions are satisfied

Design by contract (and Testing) support Traceability

- Assertions are a way to record requirements in the source code
- (Regression) tests map assertions back to the requirements

Summary(i)

You should know the answers to these questions

- What is the distinction between Testing and Design by Contract? Why are they complementary techniques?
- What's the weakest possible condition in logic terms? And the strongest?
- If you have to implement an operation on a class, would you prefer weak or strong conditions for pre- and postcondition? And what about the class invariant?
- If a subclass overrides an operation, what is it allowed to do with the pre- and postcondition? And what about the class invariant?
- Compare Testing and Design by contract using the criteria "Correctness" and "Traceability".
- What's the Liskov substitution principle? Why is it important in OO development?
- When is a pre-condition reasonable?

You should be able to complete the following tasks

- What would be the pre- and post-conditions for the methods top and isEmpty in the Stack specification? How would I extend the contract if I added a method size to the Stack interface?
- Apply design by contract on a class Rectangle, with operations move() and resize().

Summary(ii)

Can you answer the following questions?

- Why are redundant checks not a good way to support Design by Contract?
- You're a project manager for a weather forecasting system, where performance is a real issue. Set-up some guidelines concerning assertion monitoring and argue your choice.
- If you have to buy a class from an outsourcer in India, would you prefer a strong precondition over a weak one? And what about the postcondition?
- Do you feel that design by contract yields software systems that are defect free ? If you do, argue why. If you don't, argue why it is still useful.
- How can you ensure the quality of the pre- and postconditions ?