Design by Contract — Principles

Applying Design by Contract

- OO – reuse
  - potential consequence of incorrect behavior
- Reliability
- Methodological principles to produce correct and robust software
- Systematic approach to deal with abnormal cases – exception handling
- Handling inheritance through subcontract

Defensive Programming

- Tool for reliability
- Protect every software module against slings and arrows of outrageous fortunes
- encourages programmers to include as many checks as possible
- do not rely on checks by the caller
- Routines should be as general as possible
  - A partial routine (one that works only if the caller ensures certain restrictive conditions at the time of the call) is considered dangerous because it might produce unwanted consequences if a caller does not abide by the rules.

- May be self defeating
  - Adding possibly redundant code “just in case” only contributes to the software’s complexity - the single worst obstacle to software quality in general and to reliability in particular.
  - The result of such blind checking is simply to introduce more software hence more sources of things that could go wrong at execution time, hence the need for more checks, and so on

- Such blind and often redundant checking causes much of the complexity and unwieldiness that often characterizes software.
- Obtaining and guaranteeing reliability requires a more systematic approach.
- In particular, software elements should be considered as implementations meant to satisfy well-understood specifications, not as arbitrary executable texts. This is where the contract theory comes in.

Contract involving two parties

- Benefits and obligations
- each party expects some benefits from the contract
- Each party is prepared to incur some obligations

Contract involving two parties

- Usually, an obligation for one party is a benefit for the other
- If the contract is exhaustive, every "obligation" entry also in a certain sense describes a "benefit" by stating that the constraints given are the only relevant ones.

Example contract

Party	Obligations	Benefits
Client	Provide letter or package of no more than 5 kgs, each dimension no more than 2 meters. Pay 100 francs.	Get package delivered to recipient in four hours or less.
Supplier	Deliver package to recipient in four hours or less.	No need to deal with deliveries too big, too heavy, or unpaid.

- Usually, an obligation for one party is a benefit for the other
- If the contract is exhaustive, every "obligation" entry also in a certain sense describes a "benefit" by stating that the constraints given are the only relevant ones.

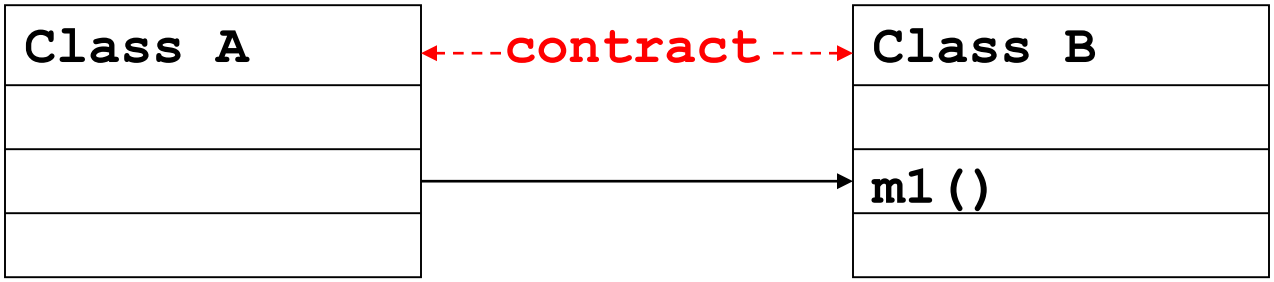
- For example, the obligation entry for the client indicates that a client who satisfies all the constraints listed is entitled to the benefits shown in the next entry.
- This is the No Hidden Clauses rule:
  - With a fully spelled out contract between honest parties, no requirement other than the contract's official obligations may be imposed on the client as a condition for obtaining the contract's official benefits.
- The No Hidden Clauses principle does not prevent us from including references. implicit or explicit, to rules not physically part of the contract

Contracting for Software

- If the execution of a certain task relies on a routine call to handle one of its subtasks, it is necessary to specify the relationship between the client (the caller) and the supplier (the called routine) as precisely as possible.
- The mechanisms for expressing such conditions are called assertions.
- Some assertions, called preconditions and postconditions, apply to individual routines. Others, the class invariants, constrain all the routines of a given class and will be discussed later.
- Each assertion is a list of boolean expression

Design by Contract

Consider the relationship between a class and its clients:



DbC views this as a formal agreement, or *contract*, expressing each party's *rights* and *obligations*.

Without such a precise definition we cannot have a significant degree of *trust* in large software systems.

Contract violations lead to run-time errors, i.e. *exceptions.*

Expressing a Specification

Let ***A*** be some operation.

A *correctness formula* is an expression of the form:

$$\{ P \}\ A\ \{ Q \}$$

P and *Q* are *assertions* (logical predicates):
P is the *pre-condition*, *Q* the *post-condition*.

It means:

"Any execution of ***A***, starting in a state where *P* holds, will terminate in a state where *Q* holds".

E.g. $\{x >= 9\}\ x := x + 5\ \{x >= 13\}$

Pre- and Post-conditions

– expresses the constraints under which a method will function properly

– applies to *all* calls of the method

– expresses properties of the state resulting from a method's execution

– expresses a guarantee that the method will yield a state satisfying certain properties, assuming it has been called with the pre-condition satisfied.

A *correct* system will *never* execute a call in a state that does not satisfy the pre-condition of the called method.

A *correct* system will *always* deliver a state that satisfies the post-condition of the called method.

Contracting for Software Reliability

p binds the *client* A. It is an *obligation* for A but a *benefit* for B.

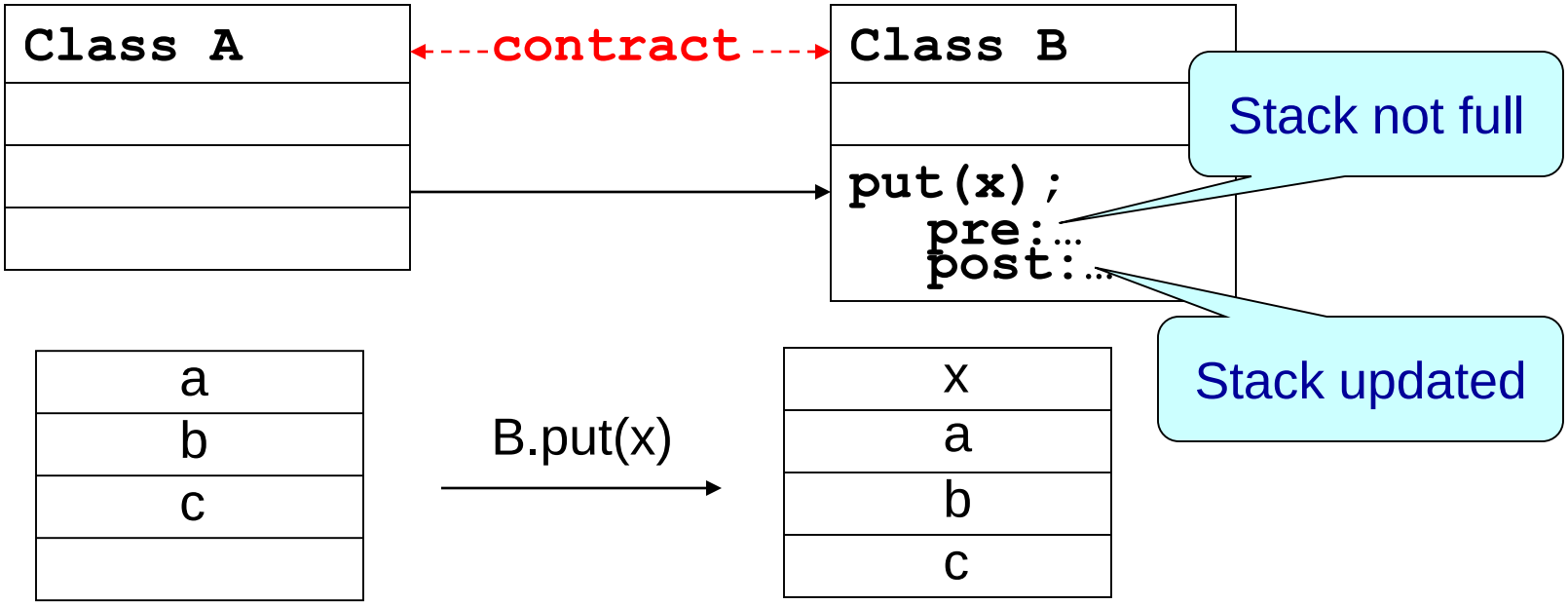
q binds the *supplier* B. It is an *obligation* for B but a *benefit* for A.

Consider

Class B says to its clients:

"If you promise to call *m* with *p* satisfied, then I, in return, promise to deliver a final state in which *q* is satisfied."

E.g. Consider the put operation for stacks:



	Obligations	Benefits
Client (**A**)	Must satisfy precondition: Only call put(x) on a non-full stack.	(From postcondition) Get stack updated.
Supplier (**B**)	Must satisfy postcondition: Update stack with x on top.	(From precondition) Simpler processing, can assume stack not full.

Handling a special case

- any runtime violation of a condition in the contract is NOT a special case; it is a bug!
- A precondition violation indicates a bug in the client – the caller did not observe the conditions imposed on correct calls.
- A post-condition violation is a bug in the supplier routine – the routine failed to deliver on its promises

Observations

- If the pre-condition is not satisfied, the routine is not bound to do anything.

- so no need to write code like:

```
if new = void then
        …
else
        …
end
```

- This is exactly opposite to the idea of defensive programming – in the presence of a contract, there is no need to have blind redundant checks.
- The stronger the pre-condition, the heavier the burden on the client and the easier for the supplier.
- the percentage of useful code to error checking code is much less in some cases.

Class Invariants

Preconditions and postconditions describe the properties of *individual methods.*

A *class invariant* is a *global* property of the *instances* of a *class*, which must be preserved by *all methods*.

A class invariant is an assertion in the class definition.

E.g. a stack class might have the following class invariant:

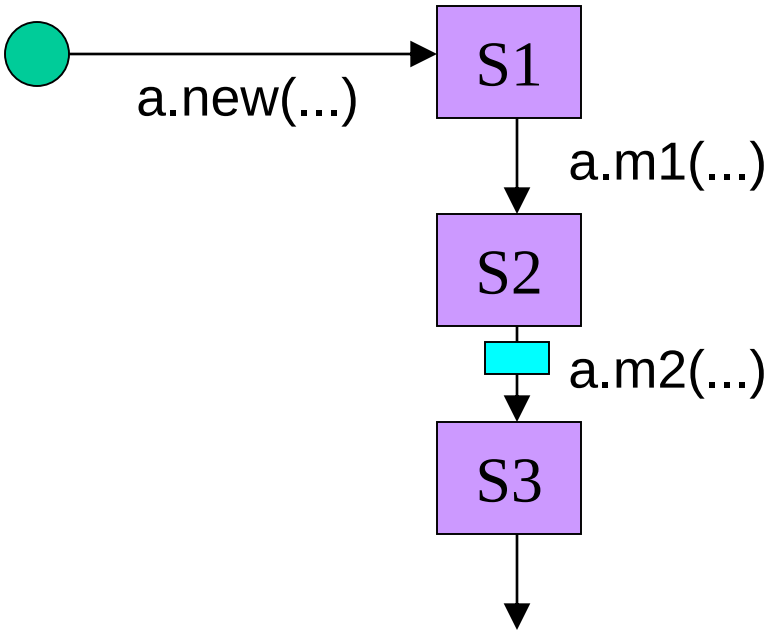
```
count >= 0   and
count <= capacity   and
stack_is_empty = (count = 0)
```

Class Stack
Attrs …
Methods …
Inv:…

An invariant may link attributes (e.g. **count** and **capacity**), or functions (e.g. **stack_is_empty**), or attributes and functions.

An invariant for a class C must be satisfied by every instance of C at all "stable" times.

"Stable" times are those in which the instance is in an observable state



i.e. on instance creation and before and after every method call.

Si is an observable state

is not an observable state

Example

E.g. a (mutable) class representing a range of real numbers:

```
public class RealRange {
private RealNumber min, max;
public RealRange(RealNumber min, RealNumber max) {
this.min = min; this.max = max;                    }
public void setRange(RealNumber newMin, RealNumber newMax)
{this.min = newMin; this.max = newMax;}
```

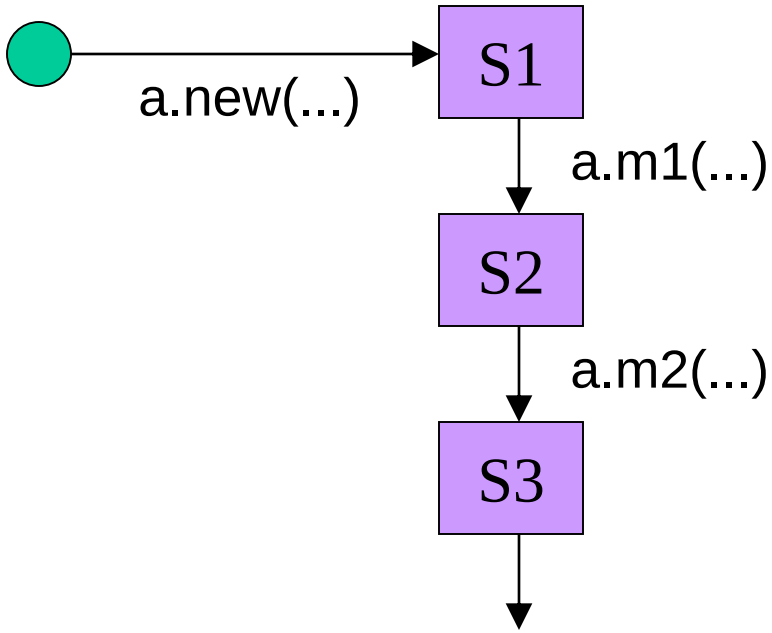
Clearly this class should have an invariant: **`min <= max.`**

- The constructor **`RealRange`** *establishes* the invariant.
- The invariant is *re-established* by the time the method **`setRange()`** *terminates*.

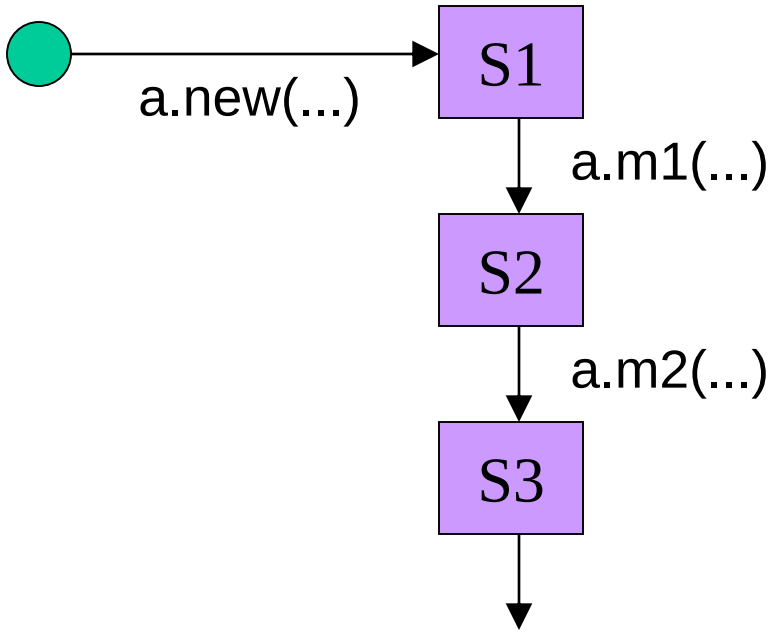
The Invariant Rule

An assertion ***I*** is a correct invariant for a class *C* if and only if:

• Every constructor of *C*, when applied to arguments satisfying its precondition in a state where the attributes have their default values, yields a state satisfying ***I***.



• Every method of the class, when applied to arguments and a state satisfying both ***I*** and the method's precondition, yields a state satisfying ***I***.



DbC and Inheritance

What happens to assertions when classes are inherited?

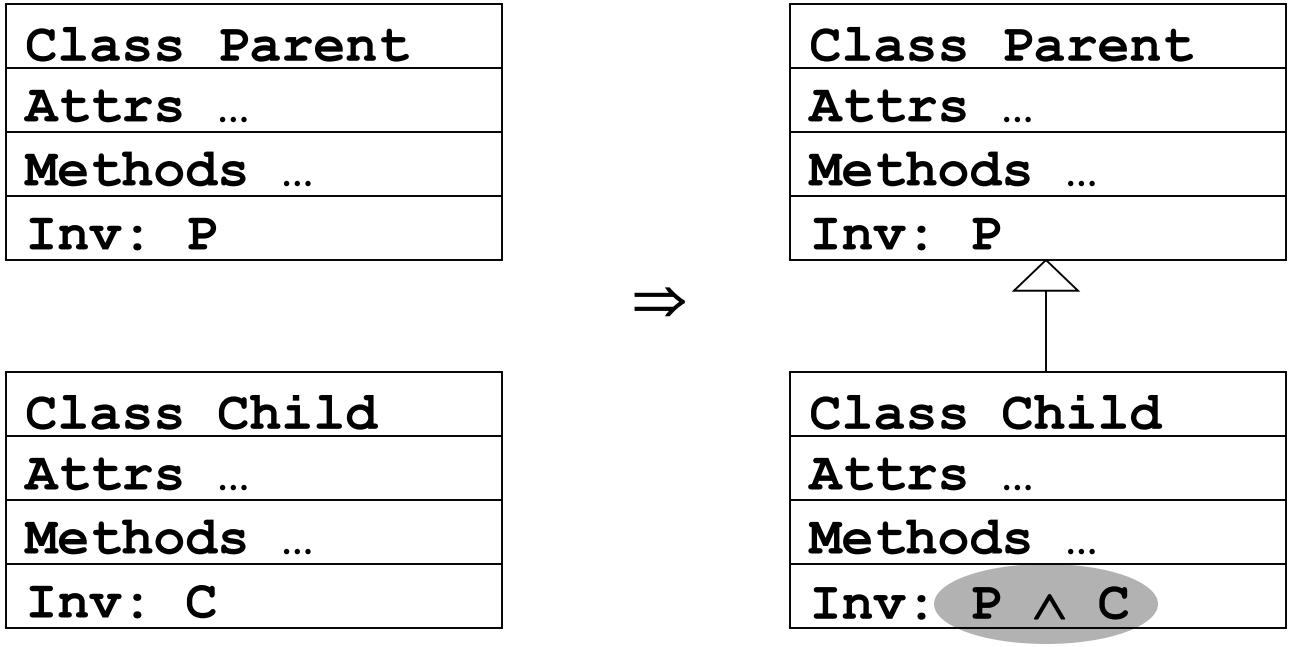
How can assertions be “preserved” in the face of redeclaration (overriding) and dynamic binding?

Actually assertions help maintain the semantics of classes and methods when they are inherited.

Invariants

The invariants of all the parents of a class apply to the class itself.

The parents' invariants are added (logically "and"ed) to the class's own invariants.



The parents' invariants need not be repeated in the class.

Pre and Postconditions

A method redeclaration may only:

replace the original *precondition* by one *equal* or *weaker*

The new version must accept all calls that were acceptable to the original.

It may, but does not have to, accept more cases.

e.g. replace **`pre: x=10`** by **`pre: x<=10`**

replace the original *postcondition* by one *equal* or *stronger*

The new version must guarantee at least as much as the original.

It may, but does not have to, guarantee more.

e.g. replace **`post: x<=10`** by **`post: x=10`**

Summary

• Software reliability requires precise specifications which are honoured by both the supplier and the client.

• DbC uses assertions (pre and postconditions, invariants) as a contract between supplier and client.

• DbC works equally well under inheritance.

Languages with third-party support:

- C and C++: DBC for C preprocessor, GNU Nana
- C#: eXtensible C# (XC#).
- Java: iContract2, Contract4J, jContractor, Jcontract, C4J, CodePro Analytix, STclass, Jass preprocessor, OVal with AspectJ, Java Modeling Language (JML), SpringContracts for the Spring framework, or Modern Jass, Custos using AspectJ.
- JavaScript: Cerny.js or ecmaDebug.
- Common Lisp: the macro facility or the CLOS metaobject protocol.
- Scheme: the PLT Scheme extension
- Perl: the CPAN modules Class::Contract , Carp::Datum
- Python, PyDBC , Contracts for Python.
- Ruby: Ruby DBC , ruby-contract.