Design by Contract

Building Reliable Software

Contracts and Quality Assurance

Contracts enable QA activities to be based on a precise description of what they expect.

Profoundly transform the activities of testing, debugging and maintenance.

"I believe that the use of Eiffel-like module contracts is the most important non-practice in software world today. By that I mean there is no other candidate practice presently being urged upon us that has greater capacity to improve the quality of software produced. ... This sort of contract mechanism is the sine-qua-non of sensible software reuse. "

Tom de Marco, IEEE Computer, 1997

Slide 228 from Bertrand Meyer

Software Correctness Property

- Correctness is a relative notion
  - » **A program is correct with respect to its specification**
  - » **To be correct a program must correspond with its specification**
    - > **print("Hello world")**
      - **Is neither correct nor incorrect**

- Correspondence is one of the cornerstones of building reliable software
  - » **Viewing the system – or part of it – from different perspectives without contradiction**

Correctness Formulae

{ Pre } A { Post }

- **A** is some operation
  - » **One or more program text statements, including procedure calls**
- **Pre** & **Post** are the preconditions and postconditions for the operation

Any execution of A, starting in a state where Pre holds, will terminate in a state where Post holds

{ P } A { Q }

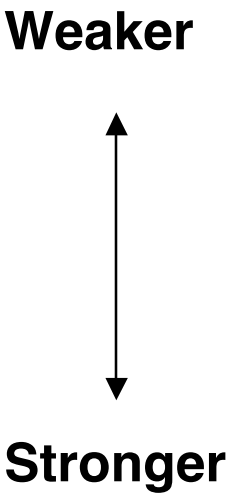
- Also called **Hoare triples**
  - » **Mathematical notation**
  - » **P & Q are assertions**
- Example

  { x ≥ 9 } x : = x + 5 { x ≥ 13 }

  > **Note assertions only need to be true**
  > **Can assertions be good? the best?**
  > **What do we mean by good with respect to assertions?**

Weak and Strong Assertions

- Suppose you are given a set of integers
  - » **{ 2 4 8 16 32 ... }**
- An assertion can be used to describe the integers in the set
  - » **1 – a set of some integers**
    - > **{ p: INTEGER · p }**
  - » **2 – a set of even integers**
    - > **{ p: INTEGER | p mod 2 = 0 · p }**
  - » **3 – a set of powers of two**
    - > **{ p: INTEGER · 2 ** p }**
  - » **4 – set of powers of two with positive exponent**
    - > **{ p: INTEGER | p > 0 · 2 ** p }**



Weak and Strong Assertions – 2

- The stronger the assertion the closer the description comes to specifying the actual set
- In general
  - » **Weak assertions describe bigger sets than strong assertions**
- In programming
  - » **The weaker the assertion the more cases that must be handled**
    - > **For precondition – more input states**
    - > **For postcondition – more output states**

Job Hunting

{ P } A { Q }

- Suppose you are looking for a job where you have to do **A**
- If **P** is weak you have to handle many cases, if **P** is strong you have fewer cases to handle
- What do you look for to make your job easier?
- What does the employer look for to get the most work out of you?

Strongest Precondition

{ False } A { ... }

- No input condition is acceptable
  - » **You do not have to do any work as the conditions are never right for you to do anything**
  - » **Independent of postcondition – A is never executed**
- The supplier – you – has no responsibility, do no work – **take the job !!!**
- The client – employer – has all the responsibility, has to do all the work as they get no work out of you

Weakest Precondition

{ True } A { Q }

- Any input condition is acceptable
  - » **As an employee you have to handle all possible situations to make Q true**
    - > **This is the most work on your part – if you are lazy you, stay away from this job**
    - > **The employer loves this, they get the most out of you**
- The supplier – you – does all the work and has all the responsibility – **taking the job depends upon Q**
- The client – employer – has no responsibility, does no work

Precondition Conclusions

- The stronger the precondition the better for the supplier, the worse for the client
- There is a tradeoff
- In practice
  - » **Have the weakest precondition that makes the task feasible**
    - > **Satisfy the most clients**
    - > **Supplier is able to satisfy the postcondition with reasonable effort**

Weakest Postcondition

{ ... } A { True }

- All output conditions are acceptable
  - » **You have an easy job, as anything you do is acceptable as long as you do something**
  - » **Independent of precondition – input not linked to output**
- The supplier – you – has minimum responsibility, do minimum work – **next best thing to strongest precondition**
- The client – employer – has all the responsibility, has to do all the work as they may not get any useful work out of you

Strongest Postcondition

{ ... } A { False }

- No output condition is acceptable
  - » **You have to work forever without achieving your goal, you are doomed to failure**
- The supplier – you – does all the work and has all the responsibility but never achieve anything
- The client – employer – has no responsibility, does no work but does not get anything done

Strongest postcondition is actually not good for either supplier or client

Postcondition Conclusions

- The stronger the postcondition the better for the client, the worse for the supplier
- There is a tradeoff
- In practice
  - » **Have the strongest postcondition that makes the task feasible**
    - > **Satisfy the most clients**
    - > **Supplier is able to satisfy the postcondition with reasonable effort**

Benefits & Obligations

	Obligations	Benefits
Client	from preconditions row & col are in range	from postconditions get requested element if it exists
Supplier	from postconditions return requested element, if it exists	from preconditions knows row and col are in range

Get more – check less

- Less programming – **Non Redundancy Principle**
  - » **Under no circumstances shall the body of a routine ever test for the routine's precondition**
  - » **Redundancy leads**
    - > **software bloat**
      - – **both size & execution time**
    - > **complexity**
    - > **more sources of error**
- Clearly indicate who has what responsibility
  - » **supplier**
  - » **client**

??? Defensive Programming ???

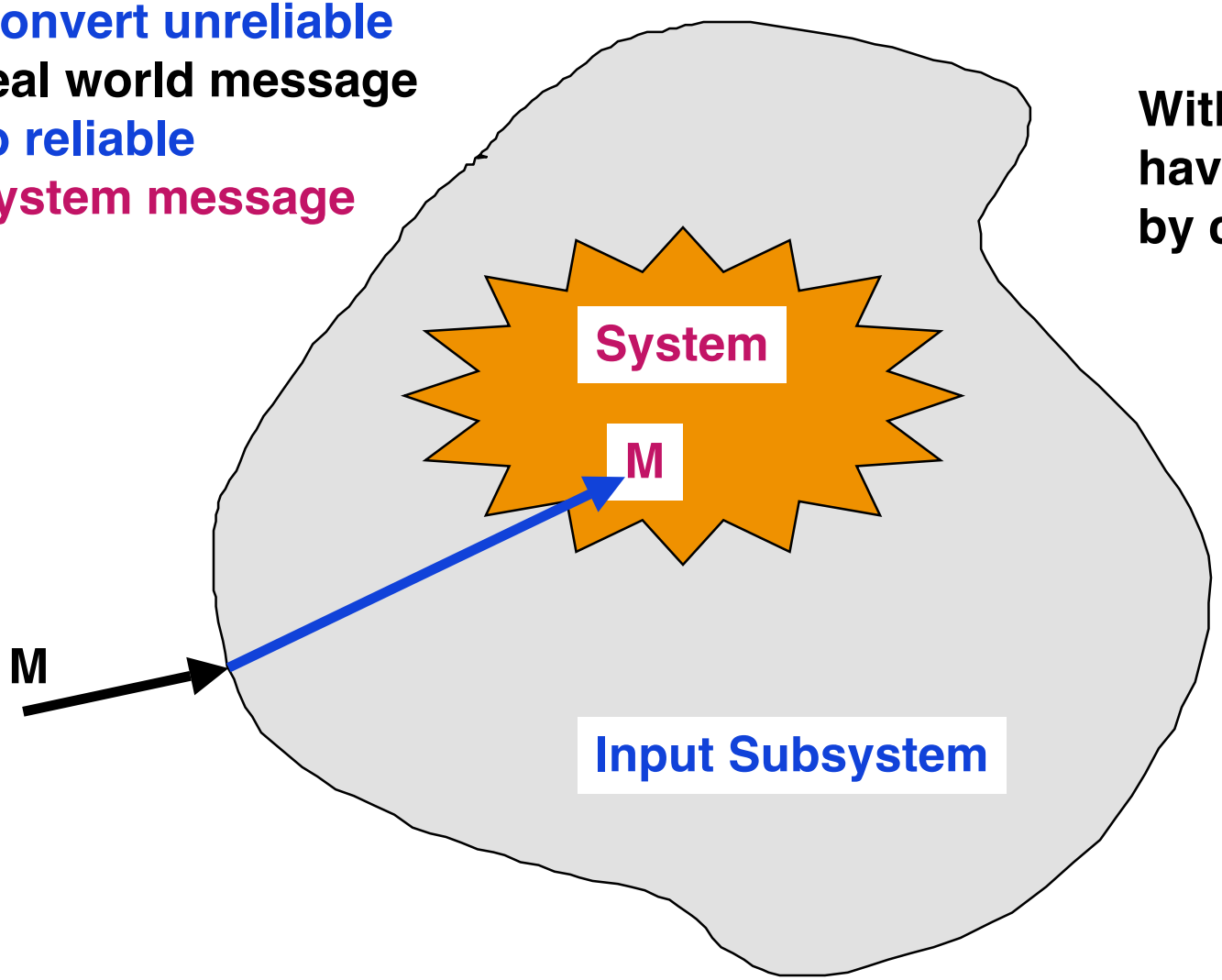
- Opposite of design by contract
  - » **Every routine checks its input irregardless of preconditions**
    - > **Effectively precondition is the weakest – True**
- Every one is responsible
  ==> No one accepts responsibility
  - » **Can always point to someone else**
- Need the notion of
  - » **The buck stops here**
- Defensive programming is undefendable

Not Input Checking

- Contracts are about **software <-> software communication**
  - » **NOT the following**
    - > **software <-> human**
    - > **software <-> real world**
- Example input routine
  - » **require: numeric key to be pressed**
    - > **Wishful thinking – cannot guarantee person will only press numeric key**
    - > **Not a contract**
  - » **Can only expect any key may be pressed**

Input Subsystem

Convert unreliable
real world message
to reliable
system message



Within system
have design
by contract

Assertion Violation Rules

- Rule 1
  - » **A run time assertion violation is the manifestation of a bug in the software**
- Rule 2
  - » **A precondition violation is the manifestation of a bug in the client**
  - » **A postcondition violation is the manifestation of a bug in the supplier**

Definitions

- Error
  - » **A wrong decision made during software development**
- Defect – bug sometimes means this
  - – **The term Fault is also used**
  - » **Property of software that may cause the system to deviate from its intended behaviour**
- Fault – bug sometimes means this
  - – **The term Failure is also used**
  - » **The event in which software deviates from its intended behaviour**

Error ==> Defect ==> Fault

Error ==> Fault ==> Failure

Imperative vs Applicative

```
full : BOOLEAN is
do
   Result := ( count = capacity )
ensure Result = (count = capacity )
end
```

- Not redundant
  - » Body is imperative – a description of how
    - > Computing – changes state
  - » Ensure is applicative – a description of what
    - > Mathematics – does not change state, either true or false

Imperative vs Applicative – 2

- Alternate bodies are possible

```
if count = capacity then Result := True
                    else Result := false end
```

```
if count = capacity then Result := True end
```

Terminology

Computing		Mathematics
Implementation	⟷	Specification
Instruction	⟷	Expression
How	⟷	What
Imperative	⟷	Applicative
Prescription	⟷	Description

Reasonable Preconditions

- Preconditions appear in the official documentation given to clients
- Possible to justify the need for the preconditions in terms of the specification only
- Every feature appearing in preconditions are available to every client to which the feature is available
  - » **No surprises**

Correctness of a Class

- A class C is correct with respect to its assertions if and only if

  For any valid set of arguments A_p to a creation procedure P

C1 **{Def C and pre P (A_P) } Body P {post P (A_P) and inv}**

- Where

  Def C assert attributes of C have default values

  pre P are the preconditions of P

  post P are the postconditions of P

  inv are the class invariants

Correctness of a Class – 2

For every exported routine R and any set of valid arguments A_R

C2 { $pre_R(A_R)$ and inv } $Body_R$ { $post_R(A_R)$ and inv}

Contract Guidelines – Class Invariant

- Develop first
- Show invariant properties of individual attributes
- Show as many invariant relationships among the attributes as possible
- Most important to show the important and non-obvious relationships
  - » **Even if it means some redundancy**
  - » **Point is not to give the logical minimum but to convey information to all clients (both developers and users)**
- As contracts for routines are developed consider general cases that may be put into class invariants

Contract Guidelines – Precondition

- Parameter-less functions can be called at any time
  - » **Precondition is always true**
    - > **As a consequence, redundant to state**
- Parameter-less procedures may have preconditions on the state or may not
  - » **As a consequence, must always assert a precondition, even if the assertion is "True"**
- Routines with parameters typically have conditions on the parameters and on the state
  - » **As a consequence, must always assert a precondition, even if the assertion is "True"**

Contract Guidelines – Precondition – 2

- Give the weakest reasonable precondition
  - » **The routines will be most useful to clients**
- All features in the precondition must be exported to the client
  - » **They must be able to execute the precondition to be sure that it is true before calling the routine.**
- Class invariants are implicitly a part of the precondition
  - » **But the client is not responsible for satisfying them**
    - > **that is a responsibility of the supplier**

Contract Guidelines – Postcondition

- Postconditions involve the all the parameters and state
  - » **Consider all possible relationships**
  - » **Specify everything that changes**
  - » **Specify everything that does not change**
    - > **Default: if no change is not mentioned, then arbitrary change, including no change, is permitted**
- For functions
  - » **Must precisely specify the value of Result**

Contract Guidelines – Postcondition – 2

- Give the strongest reasonable postcondition
  - » **Most informative to clients**
- Class invariants are implicitly a part of the postcondition
  - » **Normally not repeated but in important and non-obvious cases redundancy may be good to have**
    - > **Particularly important if there are many class invariants and only one or two apply that may be forgotten.**

Contract Guidelines – Postcondition – 3

- Features in the postcondition do not need to be exported to the client
  - » **Clients do not execute postconditions**
  - » **Some postconditions are implementation dependent**
    - > **Developer wants to make sure the implementation is correct – must be able to reference non-exported features**
  - » **But will involve some exported features as clients need to understand what the routine does.**

Contract Guidelines

- Contracts are the equivalent of security
  - » **Need to think of how security could be broken and prevent it**
- Cannot specify everything
  - » **Too much to specify**
    - > **Need to leave some things to good practice**
      - – **E.g. non-change is often left as a comment, as formal specification can be too cumbersome and non-change is common practice in the given context**
  - » **Concentrate on**
    - > **most important assertions**
    - > **non-obvious assertions**