Inheritance and Design by Contract

Parents Invariant Rule

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

Parents Invariant Rule – 2

- The invariants of all the parents of a class apply to the class itself
  - » **The parent's invariants are AND'ed together, along with the invariants of this class**

Parents Invariant Rule – 3

- The invariants of all the parents of a class apply to the class itself
  - » **The parent's invariants are AND'ed together, along with the invariants of this class**
  - » **If no invariants are given then TRUE is assumed**

Parents Invariant Rule – 4

- The invariants of all the parents of a class apply to the class itself
  - » **The parent's invariants are AND'ed together, along with the invariants of this class**
  - » **If no invariants are given then TRUE is assumed**
- Flat and interface forms provide a convenient way to see the whole story

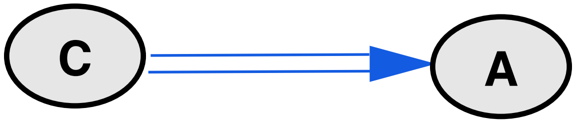
Parents Invariant Rule – 5

- The invariants of all the parents of a class apply to the class itself
  - » **The parent's invariants are AND'ed together, along with the invariants of this class**
  - » **If no invariants are given then TRUE is assumed**
- Flat and interface forms provide a convenient way to see the whole story
  - » **Flat is used by the supplier**
  - » **Interface is used by the client**

Parents Invariant Rule – 6

- The invariants of all the parents of a class apply to the class itself
  - » **The parent's invariants are AND'ed together, along with the invariants of this class**
  - » **If no invariants are given then TRUE is assumed**
- Flat and interface forms provide a convenient way to see the whole story
  - » **Flat is used by the supplier**
  - » **Interface is used by the client**
    - > **Does not have class history – redefine, rename, etc.**

Meaning of Design by Contract



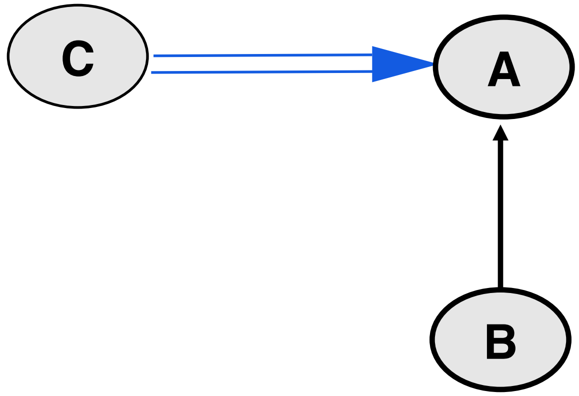
r **require** α
...
ensure β
end

Verify preconditions
if not clear they are satisfied

-- In C
a1 : A
if a1.α then
    a1.r
    check a1.β
    ... assume a1.β is true
end

Verify postconditions.
Not needed with exception handling

Enter Dynamic Binding



r require α
...
ensure β
end

r++ require γ
...
ensure δ
end

```
-- In C
a1 : A
a1 := instance of type B
if  a1. ?pre? then
    a1.r
    check a1. ?post?
    ... assume a1. ?post? is true
end
```

What are ?pre? and ?post?

What restrictions are on γ and δ ?

How to cheat

```
-- In C
a1 : A
a1 := instance of type B
if  a1. ?pre? then
    a1.r
    check a1. ?post?
    ... assume a1. ?post?
end
```

- Two ways
  - » C expects α is sufficient but B has stronger preconditions
    - > don't accept all inputs
    - > demand more from client
    - > client is wrong

How to cheat – 2

```
-- In C
a1 : A
a1 := instance of type B
if  a1. ?pre? then
    a1.r
    check a1. ?post?
    ... assume a1. ?post?
end
```

- Two ways
  - » C expects α is sufficient but B has stronger preconditions
    - > don't accept all inputs
    - > demand more from client
    - > client is wrong
  - » C expects β is delivered but B has weaker postcondition
    - > deliver outside the range
    - > effectively deliver less

Be Honest

- Replace precondition with a weaker precondition
  - » **Expect less from the client than they are prepared to do**
    - > **require clause becomes weaker**

Be Honest – 2

- Replace precondition with a weaker precondition
  - » **Expect less from the client than they are prepared to do**
    - > **require clause becomes weaker**

- Replace postcondition with a stronger postcondition
  - » **Deliver more to the client than they expect to get**
    - > **ensure clause becomes stronger**

Be Honest – 3

- Replace precondition with a weaker precondition
  - » **Expect less from the client than they are prepared to do**
    - > **require clause becomes weaker**

- Replace postcondition with a stronger postcondition
  - » **Deliver more to the client than they expect to get**
    - > **ensure clause becomes stronger**

- Willing to do the job as good as or better

Design by Contract with Dynamic Binding

- Contracts cannot be broken by redefinition

DbC with Dynamic Binding – 2

- Contracts cannot be broken by redefinition
- Assertions require and ensure are inherited

DbC with Dynamic Binding – 3

- Contracts cannot be broken by redefinition
- Assertions require and ensure are inherited
  - » **Every behaviour of the redefined method satisfies the original contract**

DbC with Dynamic Binding – 4

- Contracts cannot be broken by redefinition
- Assertions require and ensure are inherited
  - » **Every behaviour of the redefined method satisfies the original contract**
  - » **But can do more**

DbC with Dynamic Binding – 5

- Contracts cannot be broken by redefinition
- Assertions require and ensure are inherited
  - » **Every behaviour of the redefined method satisfies the original contract**
  - » **But can do more**
    - > **Accept more input cases**
    - > **Deliver more specific outputs**

Subcontracting

- Redefinition is like subcontracting

Subcontracting – 2

- Redefinition is like subcontracting
- To validate a subcontract requires a theorem prover for the general case

$$\alpha \rightarrow \gamma \qquad \textbf{and} \qquad \beta \rightarrow \delta$$

Subcontracting – 3

- Redefinition is like subcontracting
- To validate a subcontract requires a theorem prover for the general case

$$\alpha \rightarrow \gamma \quad \textbf{and} \quad \beta \rightarrow \delta$$

- This is inefficient so we provide an approximation

Subcontracting – 4

- Redefinition is like subcontracting
- To validate a subcontract requires a theorem prover for the general case

$$\alpha \rightarrow \gamma \qquad \textbf{and} \qquad \beta \rightarrow \delta$$

- This is inefficient so we provide an approximation based on the following

$$\alpha \rightarrow (\ \alpha \ \textbf{or} \ \gamma\ )$$

> **Weaker precondition is to accept α or γ**

$$(\ \beta \ \textbf{and} \ \delta\ ) \rightarrow \beta$$

> **Stronger postcondition is to accept β and δ**

Subcontracting – 5

- Language support
  - » **When redefining do not use require and ensure**
  - » **Use require else** γ

    γ **is or'ed with** α **– the inherited precondition**
  - » **Use ensure then** δ

    δ **is and'ed with** β **– the inherited postcondition**

Subcontracting example

Original definition

```
invert (epsilon : REAL )        -- Invert matrix with precision epsilon
    require   epsilon >= 10^(– 6)
    ...
   ensure abs ((Current * inverse ) – Identity )  <=  epsilon
end
```

Redefinition

```
invert (epsilon : REAL )          -- Invert matrix with precision epsilon
    require else   epsilon >= 10^(– 20)
    ...
   ensure then abs ((Current * inverse ) – Identity )  <= ( epsilon / 2 )
end
```

Assertion Redeclaration Rule

- In the redeclared version of a routine it is not permitted to use a **require** or an **ensure** clause.

Assertion Redeclaration Rule – 2

- In the redeclared version of a routine it is not permitted to use a **require** or an **ensure** clause. Instead you may:
  - » **Use a clause introduced by require else to be or'ed with the original precondition**

Assertion Redeclaration Rule – 3

- In the redeclared version of a routine it is not permitted to use a **require** or an **ensure** clause. Instead you may:
  - » **Use a clause introduced by require else to be or'ed with the original precondition**
  - » **Use a clause introduced by ensure then to be and'ed with the original postcondition**

Assertion Redeclaration Rule – 4

- In the redeclared version of a routine it is not permitted to use a **require** or an **ensure** clause. Instead you may:
  - » **Use a clause introduced by require else to be or'ed with the original precondition**
  - » **Use a clause introduced by ensure then to be and'ed with the original postcondition**
- In the absence of such a clause the original is retained

Assertion Redeclaration Rule – 5

- In the redeclared version of a routine it is not permitted to use a **require** or an **ensure** clause. Instead you may:
  - » **Use a clause introduced by require else to be or'ed with the original precondition**
  - » **Use a clause introduced by ensure then to be and'ed with the original postcondition**
- In the absence of such a clause the original is retained
- The lazy evaluation (non-strict) form of **or else** and **and then** are used

Apparent Precondition Strengthening

- Consider the case of general containers that have no bounds on capacity

  List implementation

Apparent Precondition Strengthening – 2

- Consider the case of general containers that have no bounds on capacity

  List implementation

- Inherit from List but have a bounded capacity container

  Array implementation

Apparent Precondition Strengthening – 3

- Consider the case of general containers that have no bounds on capacity

  List implementation

- Inherit from List but have a bounded capacity container

  Array implementation

- It looks like original has no restrictions when using **add** but refinement has restrictions

  > **cannot add when full**

Apparent Precondition Strengthening – 4

- Actually have the following in the unbounded container

  require not full

  > **With full defined as returning false**

Apparent Precondition Strengthening – 5

- Actually have the following in the unbounded container

  require not full

  > **With full defined as returning false**

- In child redefine

  full : BOOLEAN do Result := (count = Capacity) end

Apparent Precondition Strengthening – 6

- Actually have the following in the unbounded container

  require not full

  > **With full defined as returning false**

- In child define

  full : BOOLEAN do Result := (count = Capacity) end

- In client have

  » **if not container.full then container.add(...) end**

Apparent Precondition Strengthening – 7

- Actually have the following in the unbounded container

  require not full

  > **With full defined as returning false**

- In child define

  full : BOOLEAN do Result := (count = Capacity) end

- In client have

  » **if not container.full then container.add(...) end**

- No changes **and no surprises** in the client

Apparent Precondition Strengthening – 8

- Actually have the following in the unbounded container

  require not full

  - > **With full defined as returning false**

- In child define

  full : BOOLEAN do Result := (count = Capacity) end

- In client have

  - » **if not container.full then container.add(...) end**

- No changes **and no surprises** in the client

- Use **abstract** preconditions

Redefining a function into an attribute

- Small problem here
  - » **Precondition becomes the weaker True as the value can be accessed at any time**
  - » **But attributes do not have a postcondition**
    - > **The postcondition is added to the class invariant**
    - > **Thereby ensuring the contract still holds**

```
foo : INTEGER
   require xyz > 0
   ...
    ensure Result = k + 1
end
```

```
foo : INTEGER
    ...
    invariant
       foo = k + 1
end
```

On Style

- » Functions without arguments could be attributes
- » Could have postcondition or use class invariants
  - > class invariants are the preferred style