**Object Oriented Software Design** ISYS1083/1084

**TuteLab 4 Design by Contract (DbC)**

# PART 1: Tutorial Questions

## Briefly explain the following:

* 1. Pre-condition A Boolean query across the method’s argument values and the object’s attribute values that must be true BEFORE the method executes for the method to execute correctly.
  2. Post-condition A Boolean query across the method’s argument values and the object’s attribute values (before and after execution) that must be true AFTER the method executes for the method to provide the correct behaviour.
  3. Invariant A Boolean query across the object’s attribute values that must be true AFTER every public method executes for the class to provide the correct behaviour. These are specified ONCE for the class.
  4. Specification The collection of method signatures and pre/post-conditions and invariants for a method/class/subsystem forms its specification.
  5. The ’old’ keyword. In the Eiffel language the old keyword refers to the value of an object’s attribute BEFORE the method was called (I.e. before this method potentially changed its value).
  6. ‘Result’ variable. In the Eiffel language Result refers to the return value of the method. It can be overwritten many times. This reduces potential bugs by forcing developers to have a single exit point for every method. (note: multiple entry points in a method is conducive to bugs as its no obvious whether or under what conditions lower blocks of code in the method are executed or not).

## Defensive Programming

* 1. What is defensive programming? Adding code at the top of each method that validates the input arguments to what the supplier needs in order to perform the request correctly (aka implement pre-conditions in code).
  2. Why is this good or bad? Bad because (1) the validation is repeated many times and

(2) many validations don’t need to be run after testing is complete as most bugs are removed and this is wasted compute resources.

* 1. What defensive code can be removed when using DbC? All of it. It translates to pre- conditions.

## Discuss the differences between Design by Contract and defensive programming. From lecture slide 34:

|  |  |
| --- | --- |
| **Design by Contract Defensive Programming** | |
| Contracts are **explicit** (visible to the client as part of the interface). | Contracts are **implicit** (invisible from the client as it is part of the code). |
| **Trust** between client and supplier when we turn off runtime contract checking in production. | **Suspicion** of the client by the supplier in development and production. |

|  |  |
| --- | --- |
| Supplier: Assume I’ve **got bugs** and check my own code properly. | Supplier: assume **I’m perfect** and don’t check. |
| Supplier: Specify the requirements in advance in order to **prevent bugs**. | Supplier: Wait for a contract to be violated then  **fix the bug after** the fact. |
| Supplier: Think about the requirements of an API during **design time**. | Supplier: Think about the requirements during  **development time**. |
| Supplier: When specifying contracts put yourself in the shoes of the client (**Empathy**). | Supplier: When specifying contracts protect myself only (**Psychopathy**). |

1. Blame
   1. What is the concept of “Blame” in Design by Contract? We can blame a developer for a bug based on whether a pre or post condition was violated.
   2. If a pre-condition is violated, who is to blame? Pre – blame the author of the client code.
   3. If a post-condition is violated, who is to blame? Post – blame the author of the supplier code.
   4. If an invariant is violated, who is to blame? Invariant: The authors of the supplier module – there will likely be more people contributing the whole module than contributing to one supplier method. Need to look into recent changes in source control (e.g. Git) to see which change (and therefore which developer) broke the invariant.
   5. Is this a good or bad thing? Good because it makes developers more accountable for their code correctness.

## Java.util.Set

Consider the Set<E> interface in the java.util package (from Java 8 API). Identify the pre-conditions and post-conditions for the following operations (the API documentation can help you here, especially the throwable exceptions).

* int size() returns the number of elements in the set
* boolean add(E e) adds an element to the set
* boolean remove(Object o) removes an element e from the set (note that the argument is not parameterised by type E)
* boolean contains(Object o) returns true if the object is contained in the set (again the argument is of type Object not E).

/\*\*

\* @post.condition (“Result >= 0”)

\*\*/

int size()

/\*\*

* @post.condition (“contains(e)”)
* @post.condition (“Result == old contains(e)”)

\*\*/

boolean add(E e)

/\*\*

* @post.condition (“! contains(o)”)
* @post.condition (“Result == old contains(o)”)

\*\*/

boolean remove(Object o)

/\*\*

* n/a

\*\*/

boolean contains(Object o)

Why are the last two arguments typed as Object and not E? Otherwise it would violates “hidden contract” in the generalised Collection interface.

Do contracts exist in all APIs or just ones with pre/post-conditions? All APIs have implicit contracts. The order and type of the arguments is a contract that is enforced by the compiler. Often APIs have many more subtle contracts that are never documented and/or specified. Its better to specify explicitly the contract than to leave it hidden or poorly documented.