Design By Contract

COMP2511, CSE, UNSW

Defensive Programming Vs Design by Contract

Defensive programming:

Tries to address unforeseen circumstances, in order to ensure the continuing functionality of the software element. For example, it makes the software behave in a predictable manner despite unexpected inputs or user actions.

- often used where high availability, safety or security is needed.
- results in redundant checks (both client and supplier may perform checks), more complex software for maintenance.
- difficult to locate errors, considering there is no clear demarcation of responsibilities.
- may safeguard against errors that will never be encountered, thus incurring run-time and maintenance costs.

Design by Contract:

At the design time, responsibilities are clearly assigned to different software elements, clearly documented and enforced during the development using unit testing and/or language support.

- clear demarcation of responsibilities helps prevent redundant checks, resulting in simpler code and easier maintenance.
- crashes if the required conditions are not satisfied! May not be suitable for high availability applications.

Design by Contract (DbC)

- Bertrand Meyer coined the term for his design of the Eiffel programming language (in 1986). Design by Contract (DbC) has its roots in work on formal specification, formal verification and Hoare logic.
- In business, when two parties (supplier and client) *interact* with each other, often they write and sign contracts to clarify the obligations and expectations. For example,

	Obligations	Benefits
Client	*(Must ensure precondition)*	*(May benefit from post-condition)*
	Be at the Santa Barbara airport at least 5 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.	Reach Chicago.
Supplier	*(Must ensure post-condition)*	*(May assume pre-condition)*
	Bring customer to Chicago.	No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price.

The example is from https://www.eiffel.com/values/design-by-contract/introduction/

Design by Contract (DbC)

Every software element should define a specification (or a contract) that governs its interaction with the rest of the software components.

A contract should address the following three questions:

- Pre-condition - what does the contract expect?

  If the precondition is true, it can avoid handling cases outside of the precondition.

  For example, expected argument value `(mark>=0) and (marks<=100)`.

- Post-condition - what does the contract guarantee?

  Return value(s) is guaranteed, provided the precondition is true.

  For example: correct return value representing a `grade`.

- Invariant - what does the contract maintain?

  Some values must satisfy constraints, before and after the execution (say of the method).

  For example: a value of `mark` remains between zero and 100.

Design by Contract (DbC)

A contract (precondition, post-condition and invariant) should be,

- declarative and must not include implementation details.
- as far as possible: precise, formal and verifiable.

Benefits of Design by Contract (DbC)

- Do not need to do error checking for conditions that not satisfy the preconditions!
- Prevents redundant validation tasks.
- Given the preconditions are satisfied, clients can expect the specified post-conditions.
- Responsibilities are clearly assigned, this helps in locating errors and resulting in easier code maintenance.
- Helps in cleaner and faster development.

Design by Contract (DbC) : Implementation Issues

- Some programming languages (like Eiffel) offer native support for DbC.
- Java does not have native support for DbC, there are various libraries to support DbC.
- In the absence of a native language support, unit testing is used to test the contracts (preconditions, post-conditions and invariants).
- Often preconditions, post-conditions and invariants are included in the documentation.
- As indicated earlier, contracts should be,
  - declarative and must not include implementation details.
  - as far as possible: precise, formal and verifiable.

UNSW SYDNEY

Design by Contract : Example using Eiffel

```
class DICTIONARY [ELEMENT]
feature
        put (x: ELEMENT; key: STRING) is
                    -- Insert x so that it will be retrievable
                    -- through key.
              require
                    count <= capacity
                    not key.empty
              ensure
                    has (x)
                    item (key) = x
                    count = old count + 1
              end

        ... Interface specifications of other features ...

invariant
        0 <= count
        count <= capacity
end
```

Precondition → require block

Postcondition → ensure block

invariant → invariant block

Design by Contract: Examples in Java

```
/**
@param value to calculate square root
@returns sqrt - square root of the value
@pre  value >= 0
@post value = sqrt * sqrt
*/
public double squareRoot ( double value );
```

```
/**
 * @invarient age >= 0
 */

public class Student {
```

```
/**
@param amount to be deposited into the account
@pre  amount > 0
@post  balance = old balance + amount
*/
public void deposit( double amount);
```

Pre-Conditions

- A pre-condition is a condition or predicate that must always be true just prior to the execution of some section of code
- If a precondition is violated, the effect of the section of code becomes undefined and thus may or may not carry out its intended work.
- Security problems can arise due to incorrect pre-conditions.
- Often, preconditions are included in the documentation of the affected section of code.
- Preconditions are sometimes tested using guards or assertions within the code itself, and some languages have specific syntactic constructions for testing .

- In Design by Contract, a software element can assume that preconditions are satisfied, resulting in removal of redundant error checking code.

- See the next slide for the examples.

Pre-Conditions: Examples

```java
/**
 * @pre (mark >=0) and (mark<=100)
 * @param mark
 */
public void printGradeDbC(double mark) {

    if(mark < 50 ) {
        System.out.println("Fail");
    }
    else {
        System.out.println("Pass");
    }
}
```

Incorrect behaviour if *mark* is outside the expected range

```java
/**
 * Get Student at i'th position
 * @pre i < number_of_students
 * @param i - student's position
 * @return student at i'th position
 */
public Student getStudentDbC(int i) {

    return students.get(i);

}
```

Throws runtime exception if (*i >= number_of_students*)

Design by Contract

No additional error checking for pre-conditions

```java
/**
 * @pre (mark >=0) and (mark<=100)
 * @param mark
 */
public void printGradeDefensive(double mark) {

    if( (mark < 0) || (mark > 100) ){
        System.out.println("Error");
    }

    if(mark < 50 ) {
        System.out.println("Fail");
    }
    else {
        System.out.println("Pass");
    }
}
```

Defensive Programming:

Additional error checking for pre-conditions

Pre-Conditions in Inheritance

- An implementation or redefinition (method overriding) of an inherited method must comply with the inherited contract for the method.
- Preconditions may be weakened (relaxed) in a subclass, but it must comply with the inherited contract.
- An implementation or redefinition may lessen the obligation of the client, but not increase it.
- For example,

```java
/**
 * @pre (theta >=0) and (theta <= 90)
 * @param theta  - angle to calculate trajectory
 * @return trajectory at angle theta
 */
public double calculateTrajectory(double theta) {
```

valid

X - not valid

Weaker Pre-condition

```java
/**
 * @pre (theta >=0) and (theta <= 180)
 * @param theta  - angle to calculate trajectory
 * @return trajectory at angle theta
 */
public double calculateTrajectory(double theta) {
```

Stronger Pre-condition

```java
/**
 * @pre (theta >=0) and (theta <=45)
 * @param theta  - angle to calculate trajectory
 * @return trajectory at angle theta
 */
public double calculateTrajectory(double theta) {
```

Post-Conditions

- A post-condition is a condition or predicate that must always be true just **after** the execution of some section of code
- The post-condition for any routine is a declaration of the properties which are guaranteed upon completion of the routine's execution[1] .
- Often, preconditions are included in the documentation of the affected section of code.
- Post-conditions are sometimes tested using guards or assertions within the code itself, and some languages have specific syntactic constructions for testing .
- In Design by Contract, the properties declared by the post-condition(s) are assured, provided the software element is called in a state in which its pre-condition(s) were true.

```
/**
@param value to calculate square root
@returns sqrt - square root of the value
@pre  value >= 0
@post value = sqrt * sqrt
*/
public double squareRoot ( double value );
```

[1] Meyer, Bertrand, Object-Oriented Software Construction, second edition, Prentice Hall, 1997.

Post-Conditions in Inheritance

- An implementation or redefinition (method overriding) of an inherited method must comply with the inherited contract for the method.
- Post-conditions may be strengthened (more restricted) in a subclass, but it must comply with the inherited contract.
- An implementation or redefinition (overridden method) may increase the benefits it provides to the client, but not decrease it.
- For example,
  - the original contract requires returning a set.
  - the redefinition (overridden method) returns sorted set, offering *more* benefit to a client.

Class Invariant

- The class invariant constrains the state (i.e. values of certain variables) stored in the object.
- Class invariants are established during construction and constantly maintained between calls to public methods. Methods of the class must make sure that the class invariants are satisfied / preserved.
- Within a method: code within a method may break invariants as long as the invariants are restored before a public method ends.
- Class invariants help programmers to rely on a valid state, avoiding risk of inaccurate / invalid data. Also helps in locating errors during testing.

Class invariants in Inheritance

- Class invariants are inherited, that means,
"*the invariants of all the parents of a class apply to the class itself.*" !
- A subclass can access implementation data of the parents, however, must always satisfy the invariants of all the parents – preventing invalid states!

END