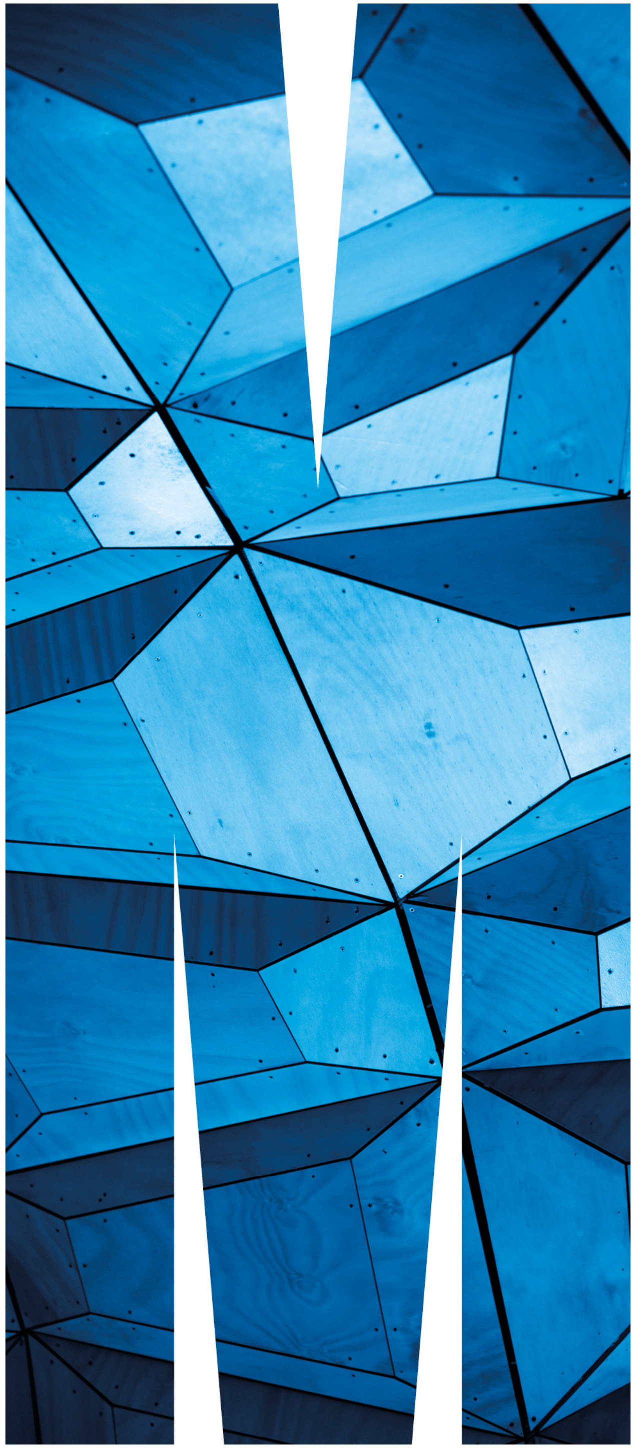
MONASH University

MONASH
INFORMATION
TECHNOLOGY

Software Specification and Design by Contract 2

FIT2099: Object-Oriented Design and Implementation

GROUP OF EIGHT AUSTRALIA



Lecture Outline

- What Happens If a Contract is Violated
  - Disciplined Exception Handling Principle
  - Non-Redundancy Principle
- Subcontracting
- Design By Contract and Inheritance
  - Liskov Substitution Principle
- Command-Query Separation

What happens if a contract is violated?

- Precondition is violated:
  - The Client/user is at fault and an exception should be thrown to the ***calling routine***
    - This means that suppliers should not try to rescue clients that have violated their preconditions
      - let the exception go to the caller − it's their fault!
      - See the Non-Redundancy Principle
    - Precondition violation is essentially always indicative of a bug – but not in the supplier

What happens if a contract is violated? ...

- Postcondition or invariant is violated:
  - The Supplier/designer is at fault and an exception is raised in the ***called routine***
  - Postcondition violation does not always indicate a bug
    - It could be due to a transient condition that prevents the method from succeeding, e.g.
      - Network outage
      - Remote server down
      - Disk or memory unavailable

Disciplined Exception Handling Principle

– Bertrand Meyer states that there are only two legitimate responses to an exception that occurs during the execution of a routine:

- ***Retrying***: an attempt to change the conditions that led to the exception and to execute the routine again from the start
- ***Failure*** (also known as organized panic): clean up the environment, terminate the call and report failure to the caller

* NB. Exceptions resulting from some operating system signals may sometimes justify a false alarm response

Exception Handling in Java

MONASH University

- In Java, a try block can be used to handle exceptions that might be thrown by code being called:

```
try {
        // code that might generate an exception
} catch (ExceptionType1 name) {
  // code to handle exceptions of type ExceptionType1
} catch (ExceptionType2 name) {
  // code to handle exceptions of type ExceptionType2
}
```

The Non-Redundancy Principle

- Bertrand Meyer states the

Non-Redundancy Principle

Under no circumstance shall the body of a routine ever test for the routine's precondition

- At first glance, this may appear to contradict "defensive programming" principles… but
  - It is the ***client's responsibility*** to check that it is meeting the preconditions of its suppliers
  - It is never a good sign for *any* code to appear twice in a program
  - A supplier cannot expect to know what it should do for all of its possible clients (some possibly not yet written)
    - A client can have code to catch and deal with an exception caused by precondition violation

Fail Fast and Design by Contract

- ‘Fail Fast’ is in fact a good principle for engineering reliable software
  - If some thing is wrong, fail immediately – let the developer know where and when the problem occurred
- Design by Contract makes it easy to write code that “fails fast”
- Code that ignores warnings is likely to fail eventually, often somewhere far in time and space from where the problem actually occurred
  - Such code is very hard to debug

Using Exceptions and Assertions

- There are four main applications of exceptions and assertions:
  - Help in writing software that is correct
  - An aid for documentation
    - Indeed, some suggest the principle “If a comment can be replaced with an assertion, do so”
  - Support for testing, debugging and quality assurance
  - Support for software fault tolerance
- Assertions are not
  - An input checking mechanism
  - Control structures

Subcontracting

- Consider the following scenario:
  - I want to get my driveway concreted
  - I find a concreter, Alice, who makes an offer to do the job.
  - We come to an agreement on what we both need to do for the job to be done to our mutual satisfaction: this is our ***contract***.

Subcontracting...

- Alice has the the following requirements:
  - – I must pay 20% of the agreed price in advance as a deposit
  - – I must make sure that the driveway gate is open on Thursday, when she can do the job
  - – I must make sure there is no car or other obstruction in the driveway on Thursday between 8:00am and 4:00pm

Subcontracting...

- These are Alice’s ***preconditions***
  – Things she needs to be true in order for her to be able to do the job.
    - She can’t lay concrete if there is a car in the way!
    - She can’t even start if the gate is locked
    - etc.

Subcontracting...

- If I do everything Alice requires, she will ensure that:
  - The concreting of driveway is completed by 4pm on Thursday
  - The newly concreted driveway will be useable by Saturday morning
  - The new driveway will be able to be used by vehicles weighing up to 2 tonnes without damage

Subcontracting...

- These are Alice’s ***postconditions***
  - Things she will ensure are done if her preconditions are met:
    - The job will be done on time
    - It will be useable in a reasonable amount of time
    - It will support the vehicles I require

Subcontracting...

- This scenario is the kind of thing we discussed in the first lecture on DbC
  – There is a contract between a supplier and a client to provide a service, with obligations and benefits on both sides
- Now, what happens if Alice wants to hire Bob to do the job instead?
  – This is subcontracting

Subcontracting...

- Under what circumstances would I by happy for Bob to do the job instead of Alice? What does Bob require?
  - Bob can’t ask for 30% deposit, we’ve already agreed on 20%!
  - Bob can’t say he can only do it on Wednesday, we’ve already agreed on Thursday!
  - Bob can’t ask for access from 5:00am, we’ve agreed on 8:00am, and I need my sleep!!!

Subcontracting...

- On the other hand, what if Bob says:
  – He’d by happy to do with only 15% paid upfront
  – He can do it on Thursday or Friday
- I’d be happy then. This is consistent with everything I agreed with Alice, and indeed a somewhat better deal
  – I’m happy to accept ***weaker*** requirements

Subcontracting...

- What about the other end of the deal? What if Bob says:
  - You won't be able to use the driveway until Sunday
  - The maximum weight the new driveway will be able to support is one tonne
- I'm not going to accept this. This is worse than what Alice agreed to provide!
  - I'm ***not*** prepared to accept weaker postconditions

Subcontracting...

- On the other hand, what if Bob says:
  - You'll be able to use the driveway from Friday 5:00pm
  - The new concrete will support vehicles weighing up to 3 tonnes
- I'm happy with this! This is everything Alice promised and more.
  - I'm happy to accept ***stronger*** postconditions

Subcontracting...

- This example shows us that a client is happy for a service to be done by a subcontractor
  - If and only if the ***preconditions are the same or weaker***
  - If and only if the ***postconditions are the same or stronger***
- Indeed, if these conditions are true, I don’t even need to know that Bob exists
  - Bob could turn up on Thursday instead of Alice, and do a job I would be completely happy with

Design By Contract and Inheritance

MONASH University

- The ***Liskov Substitution Principle*** is an important principle of good OO design in *all* languages.

Liskov Substitution Principle

- For class `S` to be a true subtype of class `T`, then `S` must conform to `T`
- A class `S` conforms to class `T`, if an object of class `S` can be provided in any context where an object of class `T` is expected and correctness is still preserved
- **In a sound OO system, all subclasses should also be true subtypes**
  - This is not always easy to achieve

Design By Contract and Inheritance…

- Application of the Liskov Substitution in the Design By Contract paradigm leads to the following rule:
  - Subclasses must honour the contracts of their parents
    - Just like Bob needed to honour the contract of Alice
  - If this is done, we can use an instance of a subclass anywhere expecting the parent class, and everything will work.

Design By Contract and Inheritance…

- The Principle of Sound Subcontracting
  - if a task is subcontracted by a supplier, the client should not need to know this
    - In particular, it should not change the requirements on the client, or the benefits the client can expect

Design By Contract and Inheritance…

MONASH University

- This leads to the following rules for preconditions and postconditions in subclasses:
  - A subclass can only ***weaken*** the preconditions of its parent(s)
    - It cannot expect more of its clients than the parent, but it can be prepared to accept less
  - A subclass can only ***strengthen*** the postconditions of its parent(s)
    - It cannot guarantee less to its clients than the parent, but it can undertake to guarantee more

Design By Contract and Inheritance…

- New preconditions of a method in a subclass should be (short-circuit) “or”ed with those of the parent. The interpretation is:
  - If the preconditions of the parent method can be met, stop there. If not try these extra (weaker) preconditions
- A method with no explicit precondition is assumed to have the precondition `True`

Design By Contract and Inheritance…

- New postconditions in a subclass should be (short-circuit) “and”ed with those of the parent. The interpretation is:
  - Meet the postconditions of the parent, and then meet these extra postconditions as well
- A feature with no explicit postcondition is assumed to have the postcondition `True`

Design By Contract and Inheritance…

- Unfortunately, it is difficult to enforce these rules automatically in a language that does not have explicit support for Design By Contract
  - Still as we have seen, extensions are available for many mainstream languages
    - Cofoja
    - Spec#
    - PyContracts
- Whether or not there is automated support, you should ***always*** obey these rules when designing class hierarchies

DbC and Inheritance Example

```
import com.google.java.contract.Requires;
import com.google.java.contract.Ensures;

public class BankAccount {

  protected int balance = 0;

  public int getBalance() {

    return balance;

  }

  @Requires({
    "amount >= 0", // must be a non-
negative amount
    "amount % 10 == 0" // only accept
deposits in multiples of 10
    })
  @Ensures({
    "balance == old (balance) + amount"
    })
```

```
  public void deposit(int amount) {

    balance += amount;

  }

  @Requires({
    "amount >= 0", // must be a non-
negative amount
    "amount <= balance" // cannot withdraw
more than is in the account
    })
  @Ensures({
    "balance == old (balance) - amount"
    })
  public void withdraw(int amount) {

    balance -= amount;

  }
}
```

DbC and Inheritance Example...

```
import com.google.java.contract.Ensures;
import com.google.java.contract.Requires;

public class OverdraftAccount extends
BankAccount {

   @Requires({
      "amount >= 0", // must be a non-
negative amount
      // Can accept deposits of any
positive amount
      })
      @Ensures({
      "balance == old (balance) + amount"
      })
    public void deposit(int amount) {

      balance += amount; // note: can't
use super due to stronger preconditions

    }
```

```
      @Requires({
      "amount >= 0", // must be a non-
negative amount
      })
      @Ensures({
      "balance == old (balance) - amount"
      // balance is allowed to go negative
      })
    public void withdraw(int amount) {

      balance -= amount;

    }
}
```

DbC and Inheritance Example…

```
public class ContractsDemo {

    public static void main(String[] args) throws Exception {

        //MeansExceptionAssertion means = new MeansExceptionAssertion();
        MeansCofoja means = new MeansCofoja(1);
        System.out.println(means.geometricMean(3, 8)); //try (-3, 8)

        BankAccount bankAccount = new BankAccount();
        bankAccount.deposit(100); // try 101
        bankAccount.withdraw(25); // try 125
        System.out.println("Balance: " + bankAccount.getBalance());

        bankAccount = new OverdraftAccount();
        bankAccount.deposit(101);
        bankAccount.withdraw(125); // try -125
        System.out.println("Balance: " + bankAccount.getBalance());
    }

}
```

Command-Query Separation

- The

Command-Query Separation Principle

software design principle states that every method should be either a command or a query

  – A ***command*** performs an action, perhaps changing the state of one or more objects
  – A ***query*** returns a value, and should have no side effects

Command-Query Separation...

- The principle of Command-Query Separation is attributed to Bertrand Meyer, the designer of Eiffel, but can be applied in any language
  - Including procedural languages. There is nothing specifically "OO" about this
  - Exceptions provide a mechanism for commands to report failure without becoming mixed with queries

Command-Query Separation…

- Command-Query Separation is particularly useful when doing Design By Contract
  - You can use any query in a precondition or postcondition check with confidence that you won’t change the state of the object that you are trying to check
- A classic example of the violation of this principle is the question “Are you awake?”

Summary

MONASH University

- Precondition violations are the responsibility of the client
- Postcondition violations are the responsibility of the supplier
- The only proper responses to an exception are to retry, or to fail
- The main body of a routine should never test for its own preconditions
- It must be possible to use an instance of a subclass wherever an object of its superclass is expected without breaking anything
- **Subclasses must honour the contracts of their parents**

References

- Meyer, B., *Object Oriented Software Construction*, Second Edition, Prentice Hall 1997, Ch. 11.
- Martin, R. *The Liskov Substitution Principle*, The C++ Report, 1996. https://drive.google.com/file/d/0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTIh/view
- Oracle, *Java Tutorial on Exceptions*, 2013. https://docs.oracle.com/javase/tutorial/essential/exceptions/