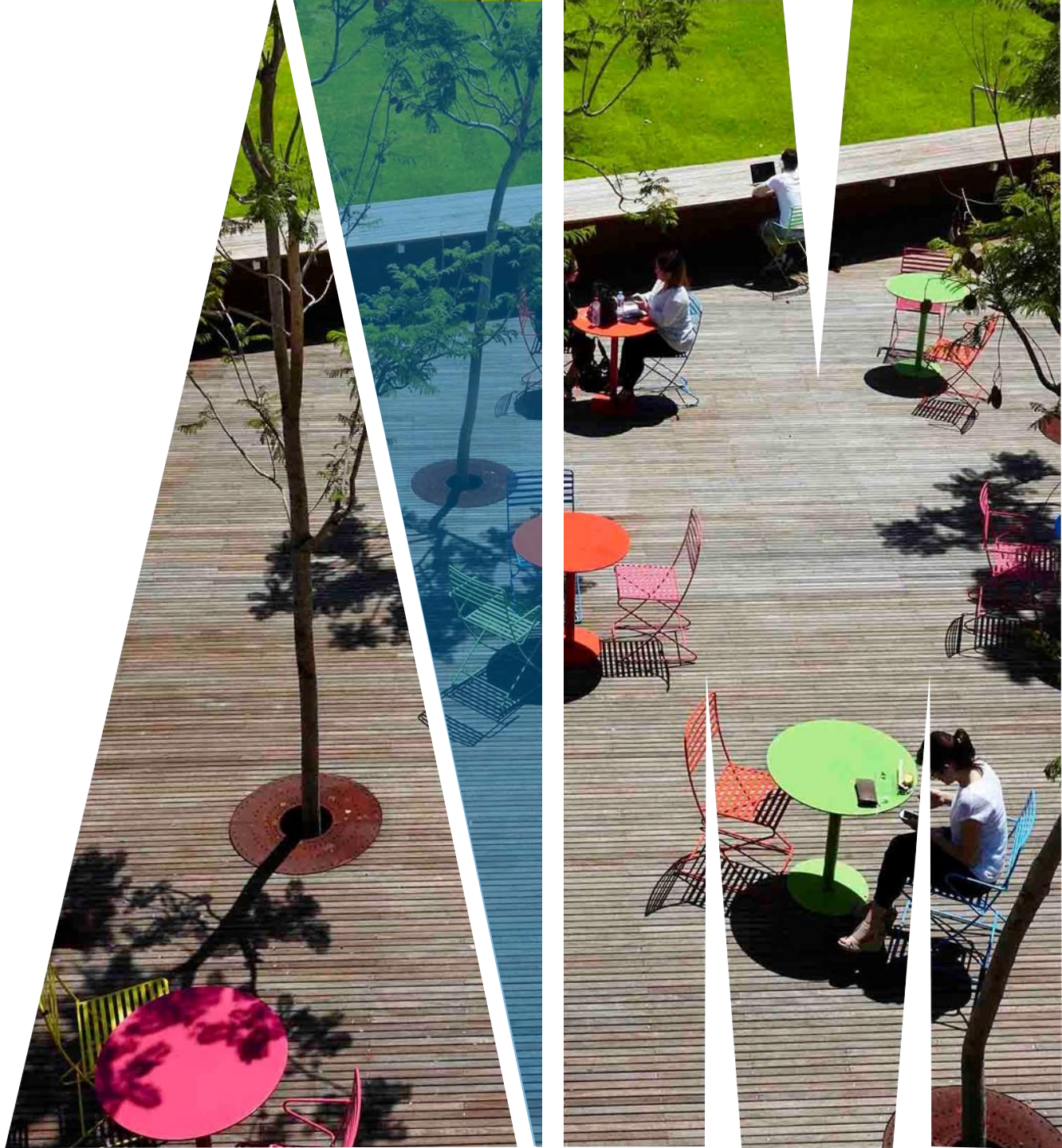
MONASH University

# FIT2099 Object-Oriented Design and Implementation

## Design by contract



MONASH University

# Outline

The basic concepts: Clients, suppliers, and contracts

Standards and contracts

Subcontracting and subclasses

Command-Query Separation principle

# WHAT IS
# DESIGN BY CONTRACT

Created by Bertrand Meyer in the 1980s, the **Design by Contract theory**, is an approach that views software design as a set of **communicating components** whose interaction is based on defined specifications of the mutual obligations — **contracts**.

The key purpose is gaining confidence that our code is correct, hence, it is:

- ❑ A systematic approach to building **bug-free object-oriented systems**.
- ❑ An effective framework for **debugging**, **testing** and, more generally, **quality assurance**.

# WHAT IS
# DESIGN BY CONTR

Created by Bertrand Meyer i is an approach that views software design as a se se interaction is based on defined specifications of t

The key purpose is gaining confiden is:

- ❑ A systematic approach to buildin
- ❑ An effective framework for **debug uality assurance**.

MONASH University

# HARDWARE VERSUS SOFTWARE DESIGN

What do hardware components have that software components (usually) lack?

Hardware components:

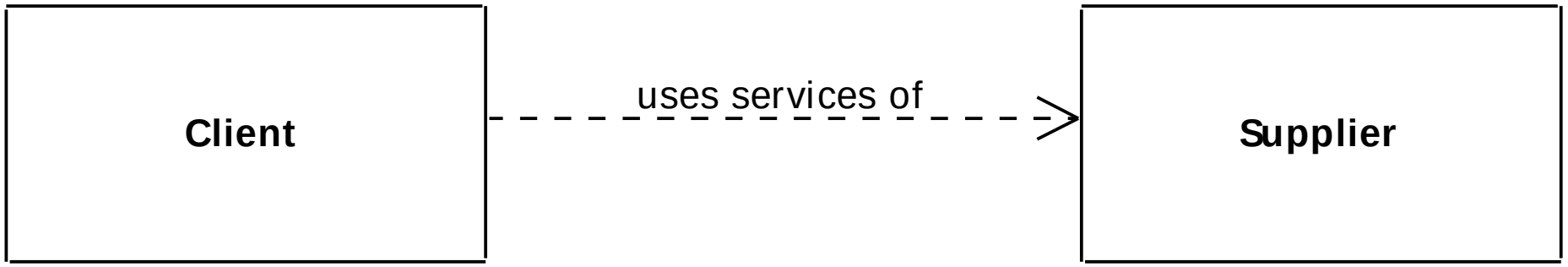
- have well-defined **public interfaces**
  - implementation is hidden (and therefore replaceable)
- have **rigorous, unambiguous specification** of behaviour
- are **well-tested**, and often guaranteed

# HARDWARE
# INTERFACES

Audio cables, light bulbs, HDMI cables, and USB cables make our lives a lot easier because they have a standardized interface (a **contract**).

In object-oriented programming, we can standardize interfaces between software modules by defining a **contract**.

# THE CLIENTS AND SUPPLIERS




A **supplier** provides services to a **client**

In UML, the client-supplier relationship is shown as an **association** or a **dependency**

- an association is used if Client has an attribute of type Supplier
- a dependency is used if it's a local variable or parameter

# DEFINING THE CONTRACT

The contract is defined by the **designer of the supplier class.**

It is communicated to the user of the class (i.e. programmer **who is writing client code**) by:

- **documentation** (e.g. Javadoc)
- **comments** in the code
- **executable statements** in the code

# THE CONTENT OF THE CONTRACT

It covers:

- **what the client needs to provide** to the class to ensure that it will operate correctly
- **what the supplier will guarantee** to be true if the class is used correctly

# EXEMPLAR METAPHOR OF
# DESIGN BY CONTRACT

The use of supplier features by a client class are governed by mutual **benefits** and **obligations**

| | OBLIGATIONS | BENEFITS |
|---|---|---|
| CLIENT | (Must ensure **preconditions**)<br>• Be at MEL airport at least 30 minutes before the departure time.<br>• Bring only allowable luggage.<br>• Pay flight ticket. | (May benefit from **postcondition**)<br>• Reach Sydney |
| SUPPLIER | (Must ensure **postcondition**)<br>• Bring client to Sydney | (May assume **precondition**)<br>• No need to carry a client who is late.<br>• No need to carry a client with unacceptable luggage<br>• No need to carry client who did not pay the ticket |

# EXEMPLAR METAPHOR OF
# DESIGN BY CONTRACT

The use of supplier features by a client class are governed by mutual **benefits** and **obligations**

| | OBLIGATIONS | BENEFITS |
|---|---|---|
| CLIENT | (Must ensure **preconditions**)<br>• Be at MEL airport at least 30 minutes before the departure time.<br>• Bring only allowable luggage.<br>• Pay flight ticket. | (May benefit from **postcondition**)<br>• Reach Sydney |
| SUPPLIER | (Must ensure **postcondition**)<br>• Bring client to Sydney | (May assume **precondition**)<br>• No need to carry a client who is late.<br>• No need to carry a client with unacceptable luggage<br>• No need to carry client who did not pay the ticket |

# THE PRECONDITIONS

The conditions that **the client must satisfy** are called **preconditions**

- the client needs to ensure that they are true before the supplier's methods are used
- if a precondition is violated, then **the client has a bug**

# THE POSTCONDITIONS

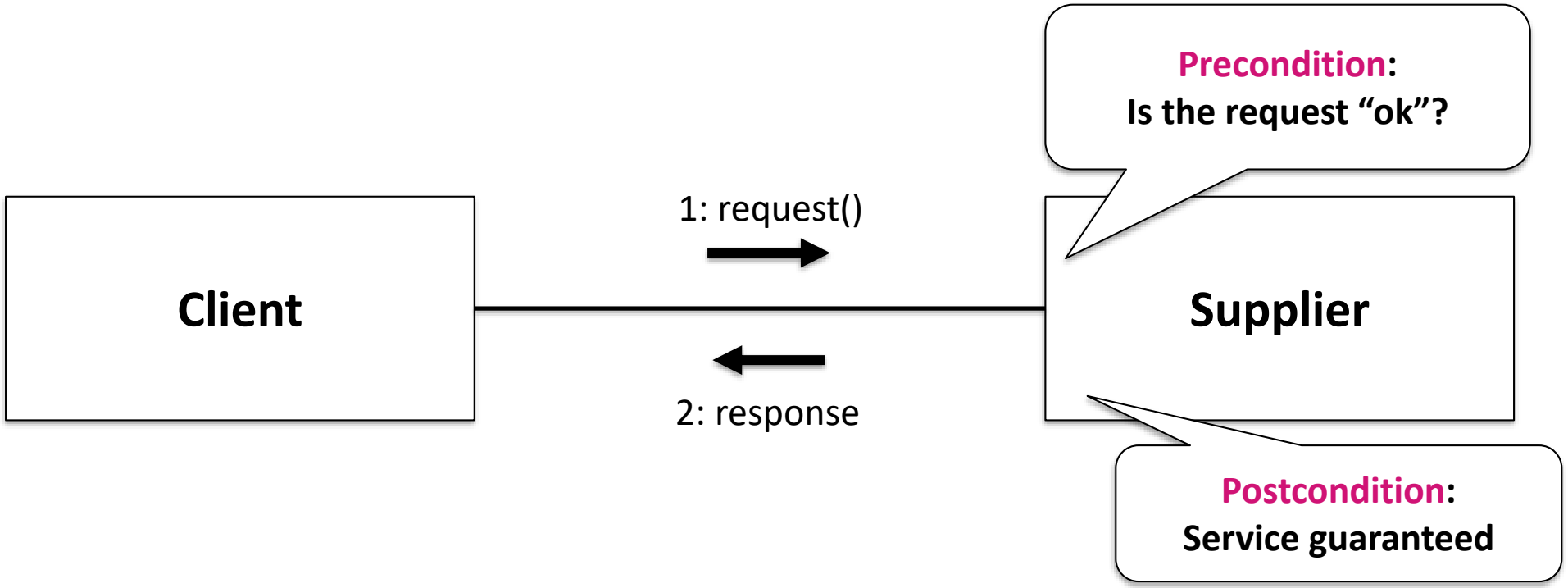
The **services that the supplier guarantees to provide** are called **postconditions**

- they describe things that are true after the supplier's method has been executed

**If the preconditions were true** when the method was called, but the postcondition has been violated, then there may be **a bug in the supplier**

- but not always; there may be an issue with (e.g.) network availability, etc.

# THE CLIENT AND SUPPLIER



# THE INVARIANT

An **invariant** is something that a class guarantees to be true at all times in order to be valid

- both before and after any supplier method is called

# FROM SPECIFICATIONS TO DESIGN BY CONTRACT

**Specifications** give us a way to think about the correctness of a class/method, and the way it is used by clients.

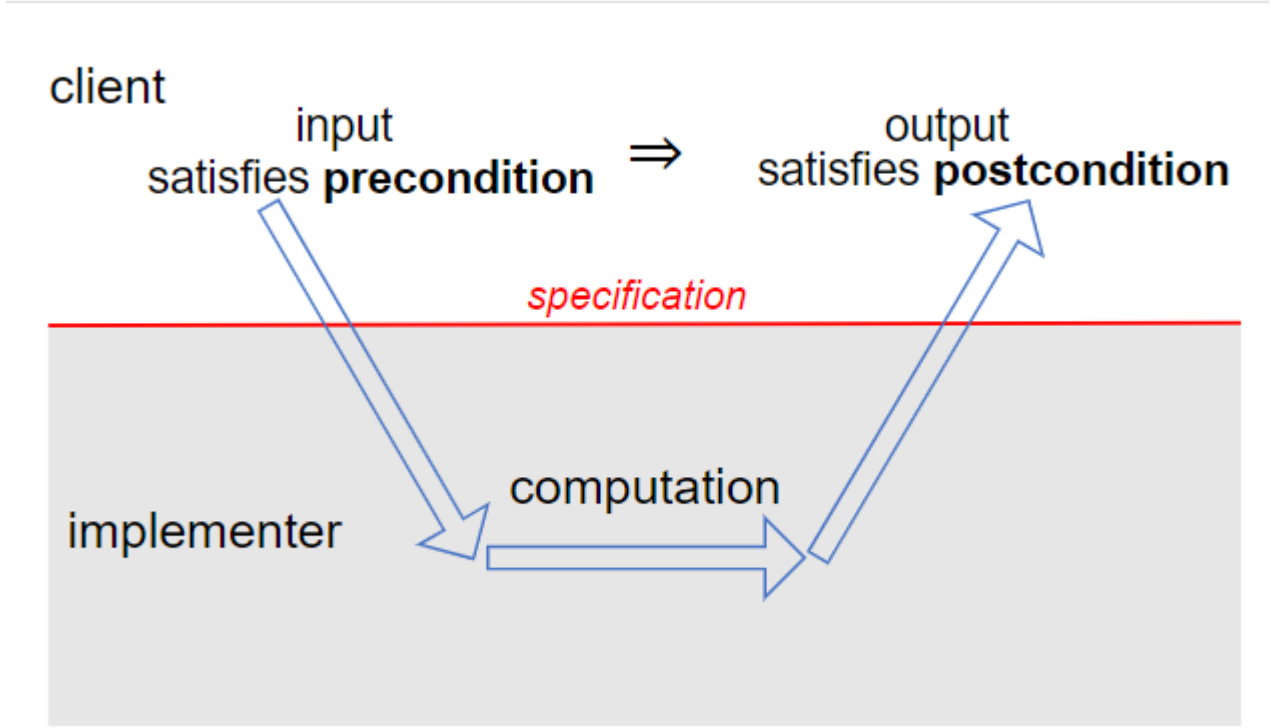
**Exception throwing** and **assertions** can be used to create **executable specifications**

Ideally, the public interface of a class is the specification, including

- **comments**
- method **signatures** (name and typed arguments)
- preconditions, postconditions, and invariants

# FROM SPECIFICATIONS TO DESIGN BY CONTRACT

**Specifications** give us a way to think about the correctness of a class/method, and the way it is used by clients.



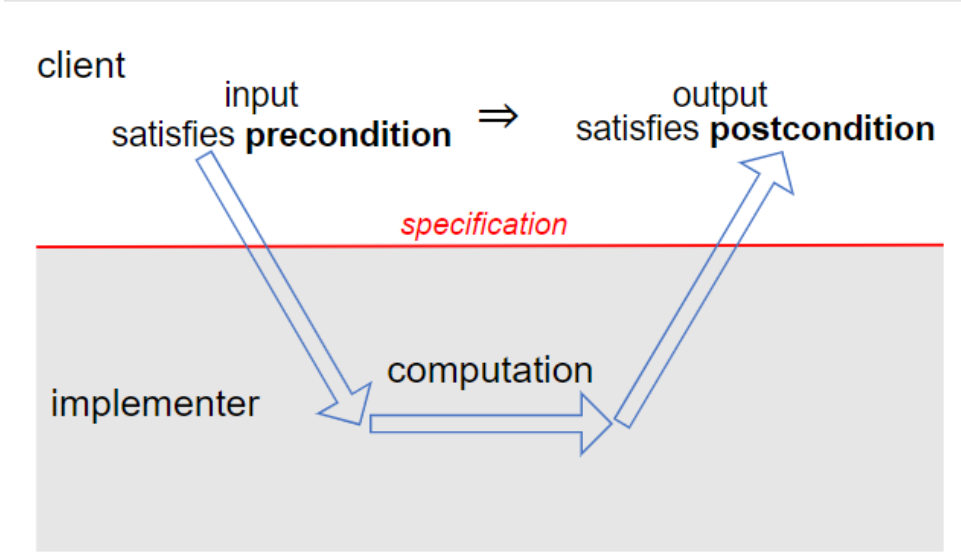

Source: mit.edu

# SPECIFICATION EXAMPLE IN JAVA

**Specification:** the method “find(value)” finds a ‘value’ in an array.

**Precondition:** the given ‘value’ occurs exactly once in the given array.

**Postcondition:** it returns the index where ‘value’ is in the array.




```
static int find(int[] array, int value)
```

Source: mit.edu

# SPECIFICATION DESCRIPTION IN THE DOCUMENTATION (JavaDoc)

**Precondition**

**Postcondition**

```
/**
 * Find a value in an array.
 * @param array array to search, requires that val occurs exactly once
 *              in arr
 * @param value value to search for
 * @return index i such that arr[i] = val
 */
static int find(int[] array, int value)
```

Source: mit.edu

# WHAT IF
# A CONTRACT IS BREACHED?

If a **precondition** is violated

- the **client** is at fault and **an exception** should be thrown to the **calling method**

  This means that suppliers should *not* try to rescue clients that have violated their preconditions

  - let the exception go back to the caller – it's their fault!
  - precondition violation always means there is a bug – but **not in the supplier**

# WHAT IF
# A CONTRACT IS BREACHED?

If a **postcondition** or **invariant** is violated

- the **supplier** is at fault
- the issue should be dealt with in the **called method**

Postcondition violation does *not* always indicate a bug

- could be due to a transient condition that prevents the method from succeeding, for example:
  - network outage
  - remote server down
  - disk or memory unavailable

# WHAT IF
# A CONTRACT IS BREACHED?

In summary:

- broken *precondition* ⇨ client's fault, throw an **exception**
- preconditions okay but broken *postcondition* ⇨ supplier's fault, throw an **assertion**

**IN JAVA:** It's okay that assertions are disabled by default in the JVM so might not be working after the code has shipped

- presumably, you'll fix the errors in your own code before you ship…

# FAIL FAST PRINCIPLE AND DESIGN BY CONTRACT

'Fail Fast' is a good principle for engineering reliable software

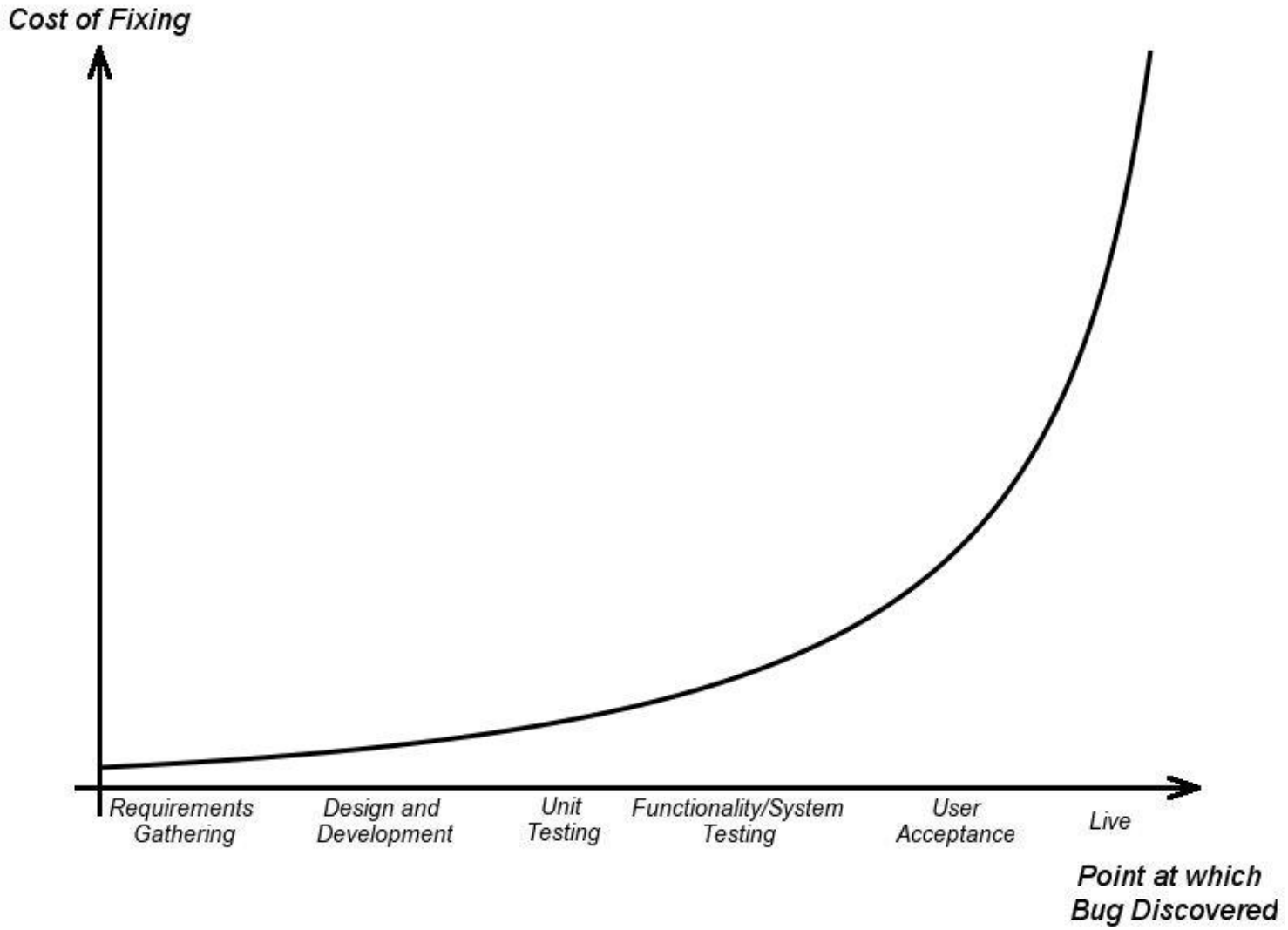
- if code detects that something is wrong, **fail immediately**
- let the developer know where and when the problem occurred

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

# FAIL FAST PRINCIPLE
# AND DESIGN BY CONTRACT



# AN INTRODUCTION TO SUBCONTRACTING

Consider the following scenario:

- I (**the client**) 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.

# AN INTRODUCTION TO 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

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.

# AN INTRODUCTION TO 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

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

# AN INTRODUCTION TO SUBCONTRACTING

So 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**

# PRECONDITIONS AND SUBCONTRACTING

Under what circumstances would I be 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!!!

I won't accept **stronger preconditions**

# PRECONDITIONS AND SUBCONTRACTING

On the other hand, what if Bob says:

- he'd by happy to do with only 15% paid upfront (instead of 20% agreed with Alice)
- 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 preconditions**

# POSTCONDITIONS AND 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 only one tonne instead of 2

I'm not going to accept this. This is worse than what Alice agreed to provide!

- I'm *not* prepared to accept **weaker postconditions**

# POSTCONDITIONS AND 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**

# SUMMARY OF 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.

# SUBCONTRACTING AND SUBCLASSES

In this scenario, **Alice** represents a **base class** and its contract, and **Bob** represents a **subclass** and its contract

- we should be able to substitute an instance of a subclass where the code expects a base class instance
- similarly, Bob can substitute for Alice if he is capable of at least fulfilling her contract

This principle is known as the **Liskov Substitution Principle** (and it is a SOLID principle that we will deeply cover next week).

# THE COMMAND-QUERY SEPARATION PRINCIPLE

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

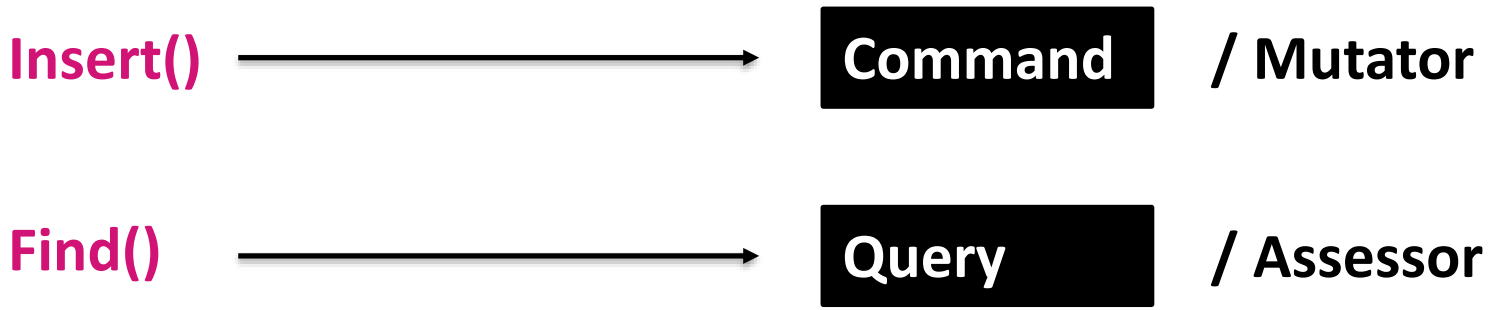
**Methods should not try to do both!**

# THE COMMAND-QUERY SEPARATION PRINCIPLE

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

The basic concepts: Clients, suppliers, and contracts

Standards and contracts

Subcontracting and subclasses

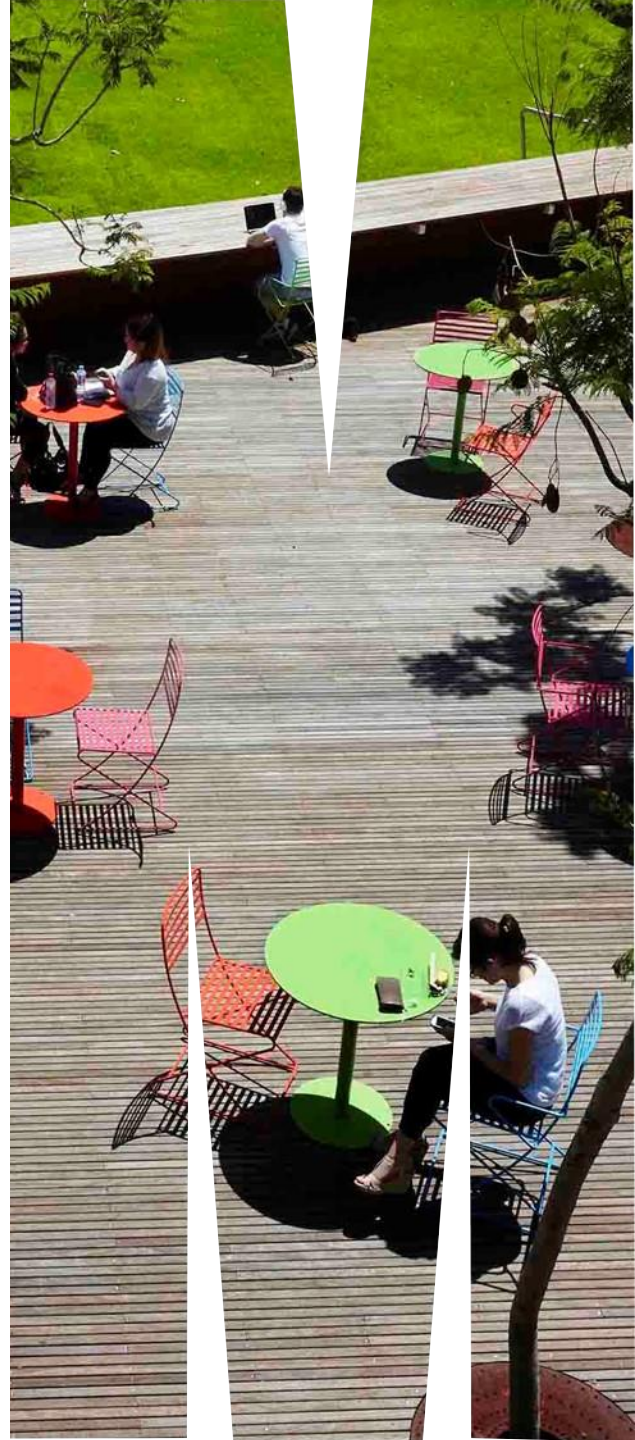
Command-Query Separation principle



Authentic example: https://cloud.google.com/speech-to-text/docs/reference/rest/v1/speech/recognize

MONASH University

# Thanks



MONASH University