Australian National University

Week 10 Announcements

- Video Assignment #2 → No late submission (hard deadline, Friday, 27 October) | 6 weeks (from the teaching break).
- Survey #2: it's open! | SELT is coming up soon!
- GitLab Repo — Lecture Code: available!
- Group Project:
  - Checkpoint 2: this week! → Registration!
  - Voice your feature: We had 15 submissions. Check feedback!
  - Surprise Feature! → Released!
- Next week: Guest lecture about IP and copyright!
- Study event: it will happen the week before the final exam (Monday, 06 November — Friday, 10 November).
- Final exam: it will be in-person in the computing labs.
  - Highly recommended to attend at least once to the labs!

COMP2100/6442
Software Design Methodologies / Software Construction

Design by Contract (DbC)

COMP2100/6442

Bernardo Pereira Nunes and
Sergio J. Rodríguez Méndez

Outline

> Overview

>> Contract

>> Interfaces

>> A bit of history

> Design by Contract

> JML

> Tools

What is a Contract?

A contract is a legally binding document that recognises and governs the rights and duties of the parties to the agreement. A contract is legally **enforceable because it meets the requirements** and approval of the law.

A contract typically involves the exchange of goods, service, money, or promise of any of those. (and in Software Development?)

"**Breach of contract**", means that the law will have to award the injured party either the access to legal remedies such as damages or cancellation. *(Wikipedia)*

(again, and in Software Development?)

Is an interface a contract?

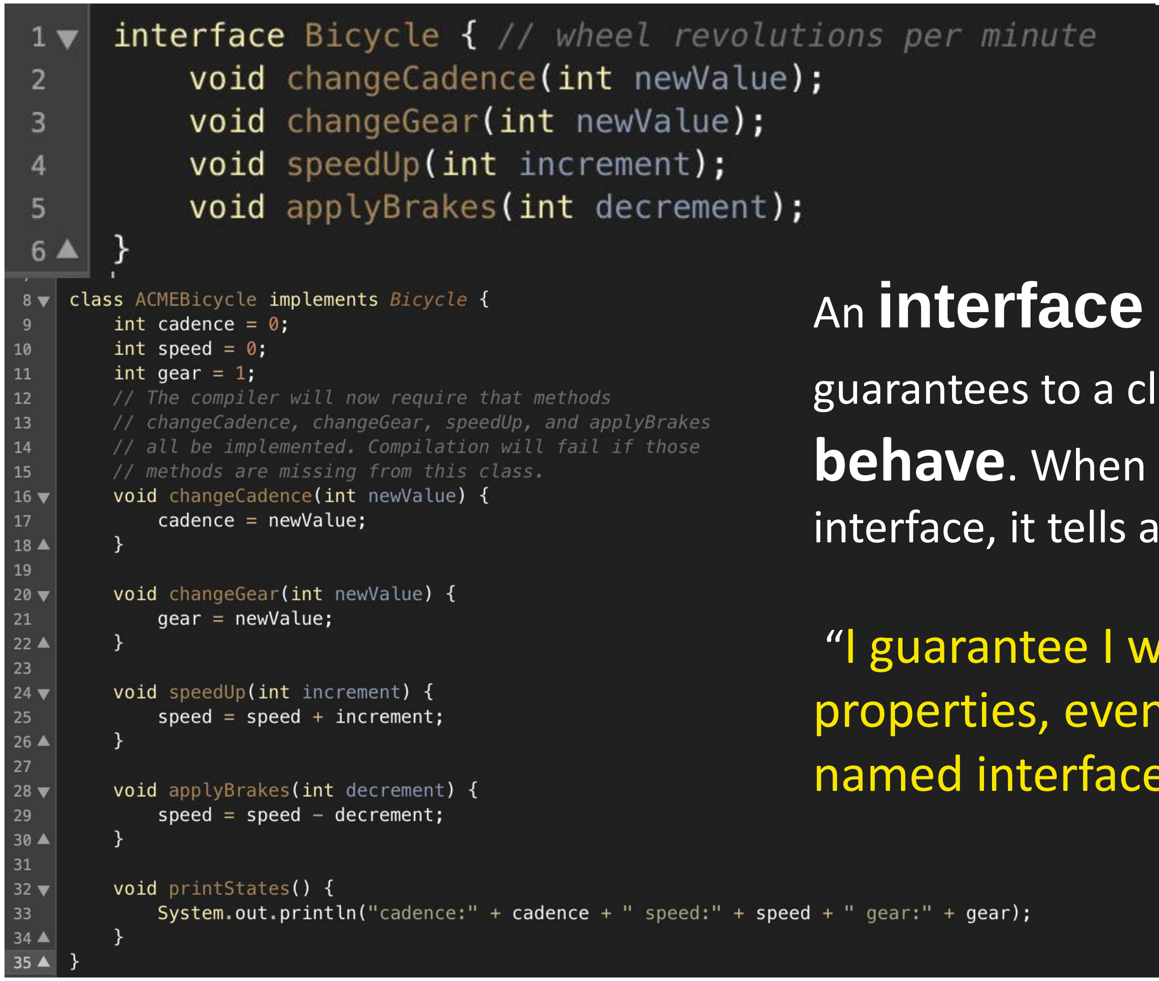
Implementing an interface allows a class to become more formal about the behaviour it promises to provide.

Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler.

If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

What is an Interface?



```java
interface Bicycle { // wheel revolutions per minute
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
```

```java
class ACMEBicycle implements Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;
    // The compiler will now require that methods
    // changeCadence, changeGear, speedUp, and applyBrakes
    // all be implemented. Compilation will fail if those
    // methods are missing from this class.
    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:" + cadence + " speed:" + speed + " gear:" + gear);
    }
}
```

An **interface is a contract** that guarantees to a client **how a class will behave**. When a class implements an interface, it tells any potential client:

"I guarantee I will support the methods, properties, events, and indexers of the named interface" .

What is Design by Contract (DbC)?

…is a method for developing software!

Idea: a class and its clients have a contract with each other.

Client: guarantees certain conditions before calling a method defined by a class (preconditions).

Class: guarantees certain properties that will hold after the call (postconditions).

A contract in software specifies both obligations and rights of clients and implementors.

A Bit of History – Design by Contract

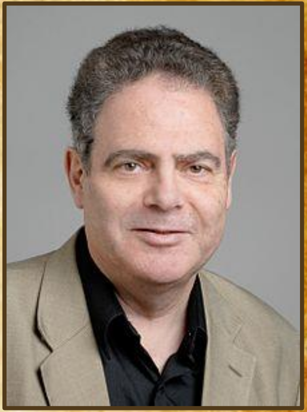
Design by Contract is a trademarked term of Bertrand Meyer.

Many developers refer to it as **Contract Programming**.

Bertrand developed DbC as part of the **Eiffel Language.**

Eiffel Language was conceived in 1985, is an object-oriented programming language and **the main goal is to increase the reliability of software construction/development**.

Many of the **concepts introduced by Eiffel Language were later adopted by other languages**, for instance, Java.


Bertrand Meyer
ETH Zurich

Basic Premise

"**To improve software reliability**, the first and perhaps most difficult problem is to define as precisely as possible, for each software element, **what it is supposed to do**".

Associate a **specification with every software element.**

These specifications (or **contracts) govern the interaction of the element with the rest of the world.**

Benefits

A **better understanding of the object-oriented method** and, more generally, of **software construction**.

A systematic approach to building "bug-free" object-oriented systems.

An effective framework for debugging, testing and, more generally, quality assurance.

A method for **documenting software components**.

Better understanding and control of the inheritance mechanism.

A technique for dealing with abnormal cases, **leading to a safe and effective language construct for exception handling**.

Example

Party	Obligations	Benefits
Client	Provide letter or package of no more than 5 kgs, each dimension < 2 meters. Pay 100 francs.	Get package delivered in 4 hours or less
UPS	Deliver package to recipient in 4 hours or less.	No need to deal with deliveries too big, too heavy, or unpaid.

Example (explanation)

The idea of DbC is a **metaphor on how elements of a software system collaborate with each other on the basis of mutual obligations and benefits.**

The metaphor comes from business life, where a "client" and a "supplier" agree on a "contract" that defines, for example, that:

- The **supplier** must provide a certain product (**obligation**) and is entitled to expect that the client has paid its fee (**benefit**).
- The **client** must pay the fee (**obligation**) and is entitled to get the product (**benefit**).

Both parties must satisfy certain obligations, such as laws and regulations, applying to all contracts.

Source: Wikipedia

Concepts: Precondition

- Preconditions specify **conditions that must hold before a method can execute**. As such, they are evaluated just before a method executes. **Preconditions involve the system state and the arguments passed into the method.**

- Preconditions specify obligations that a client of a software component must meet before it may invoke a particular method of the component. If a preconditions fails, a bug is in a software component's client.

Concepts: Postcondition

- Postconditions specify **conditions that must hold after a method completes**. Consequently, postconditions are executed after a method completes. **Postconditions involve the old system state, the new system state, the method arguments, and the method's return value.**

- Postconditions specify guarantees that a software component makes to its clients. If a postcondition is violated, the software component has a bug.

Concepts: Invariants

- An invariant specifies **a condition that must hold anytime a client could invoke an object's method.**

- Invariants are defined as part of a class definition. In practice, invariants are evaluated anytime before and after a method on any class instance executes. **A violation of an invariant may indicate a bug in either the client or the software component.**

Rationale

A contract document protects both the client, by specifying **how much should be done**, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.

The obligations of the supplier become the benefits to the client.

Assertions (pre and postconditions)

```
routine_name (argument declarations) is
    require
                Precondition
    do
                Routine body (instructions)
    ensure
                Postcondition
    end
```

Violation of an Assertion

A **precondition violation indicates** a bug **in the client (caller)**; the caller did not observe the conditions imposed on correct calls

A **postcondition violation is** a bug **in the supplier (called routine)**; the routine failed to deliver on its promises

Effect on Software

> **strong preconditions**

- heavier burden on the client
- lighter burden on the supplier

> dealing with abnormal values is a pragmatic decision about **division of labor**

Example

Obligation of the client: pass a positive number
Client has the right to: obtain the square root approximation as the result

Obligation of the supplier: compute and return a square root approximation
Supplier can assume that it will receive a positive number

Clients are obliged to establish the precondition of each method called, but in return they are saved the trouble of achieving the postconditions themselves.

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @         .approximatelyEqualTo
  @          (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

This is a contract for a **sqrt** method. It takes a number and returns its square root.

The static method *approximatelyEqualTo* of the class *JMLDouble* tests whether the relative difference of the first two double arguments is within the given epsilon, the third argument.

Tools

JML – Java Modelling Language
http://www.jmlspecs.org/

iContract – Java Design by Contract Tool

AssertMate – Java Design by Contract Tool

JASS – Java Design by Contract Tool
http://semantik.informatik.uni-oldenburg.de/~jass/

C4J – Java Design by Contract Tool
http://c4j.sourceforge.net
…

Java Modelling Language (JML)

Formal specification language for Java

> to specify behaviour of Java classes.
> to record design and implementation decisions.

by adding assertions to Java source code, e.g.,

preconditions • postconditions • invariants

The goal of JML is to be easy to use for any Java programmer

Java Modelling Language (JML)

To make JML easy to use and understand:

Properties can be specified as comments in .java source file:
/*@ . . . @*/

or

//@

*You can also use a separate file (.jml)

Properties are specified in Java syntax, namely as Java boolean expressions:

- extended with a few **operators** (\old, \forall, \result, ...).
- using a few **keywords** (requires, ensures, invariant, pure, non null, ...)

Example JML Specification

```
public class IntegerSet {

byte[] a; /* The array a is sorted */

...
```

Example JML Specification

```
public class IntegerSet { ...

byte[] a; /* The array a is sorted */

/*@ invariant
      (\forall int i; 0 <= i && i < a.length-1; a[i] < a[i+1]);
@*/


...
```

Example JML Specification

Informal comment

VS

formal JML invariant

“The array a is sorted”

```
(\forall int i; 0 <= i && i < a.length-1; a[i] < a[i+1])
```

They document the same property, but:

• JML spec has a precise meaning (e.g., < not <=)
• Precise syntax and semantics allows tool support:
    > runtime assertion checking: executing code and testing all assertions for a given set of inputs

• verification: proving that assertions are never violated, for all possible inputs

Australian National University

Running Example

```
public class BankAccount {

final static int MAX_BALANCE = 1000;
int balance;

int debit(int amount) {
    balance = balance - amount;
    return balance;
}
int credit(int amount) {
    balance = balance + amount;
    return balance;
}
public int getBalance(){
    return balance;
}
...
```

Australian National University

Requires

Precondition for method can be specified using requires:

```
/*@ requires amount >= 0; @*/
public int debit(int amount) {
    ...
}
```

Anyone calling the debit method has to guarantee the precondition.

Ensures

Postcondition for a method can be specified using ensures:

```
/*@ requires amount >= 0;
    ensures balance == \old(balance)-amount && \result == balance;
@*/
    public int debit(int amount) {
         ...
    }
```

Anyone calling the debit method can assume postcondition (if method terminates normally, i.e., does not throw exception)

\old(...) has obvious meaning

Design by Contract in JML

Pre and postcondition define a contract between a class and its clients:

- Client must ensure precondition and may assume postcondition
- Method may assume precondition and must ensure postcondition

E.g., in the example specs for the debit method, it is the obligation of the client to ensure that *amount* is positive. The requires clause makes this explicit.

Requires and Ensures

JML specifications can be as strong or as weak as you want:

```
/*@ requires amount >= 0;
      ensures true;  @*/
public int debit(int amount) {
    ...
}
```

Default postcondition “ensures true” can be omitted.
Idem for default precondition “requires true”.

(* text here *) JML treats informal description as a Boolean expression.

You can combine informal and formal descriptions (mainly when the formal statements are not easy/clear to write or expensive).

```
//@ requires (* x is positive *);
/*@ ensures (* \result is an
  @             approximation to
  @             the square root of x *)
  @       && \result >= 0;
  @*/
public static double sqrt(double x) {
  return Math.sqrt(x);
}
```

Invariants

Invariants (aka class invariants) are properties that must be maintained by all methods!

For example:

```
public class BankAccount {
        final static int MAX_BALANCE = 1000;
        int balance;
        /*@ invariant 0 <= balance && balance <= MAX_BALANCE; @*/
…}
```

> Invariants are implicitly included in all pre and postconditions
> Invariants must be **preserved** if exception is thrown!
> Invariants must be **true at the end of constructor**
> Invariants must be **preserved by each method**
> Invariants *may* be violated during the method execution

Nullable & Non_Null

```
/*@ non_null @*/ int[] a;
```

As most references are non-null, JML takes this as default.

Only nullable fields need to be annotated, e.g.

```
/*@ nullable @*/ int[] b;
```

Default is non_null (!)

Pure (also called *query* methods)

Methods without side-effects that are guaranteed to terminate or throw an exception can be declared as pure:

```
/*@ pure @*/ int getBalance () {
    return balance;
}
```

Pure methods can be used in JML annotations:

```
//@ requires amount <= getBalance();
public void debit (int amount){
    ...
}
```

```
*Java's assignment expressions are not allowed (=, +=, -=,
…, --, ++). Only pure methods can be called in assertions.
```

Assignable

For non-pure methods, class properties can be modified using assignable clauses, for example:

```
/*@ requires amount >= 0 && balance >= amount;

 assignable balance;

ensures balance == \old(balance) - amount; @*/
public void debit(int amount){
    ...
}
```

debit is only allowed to modify the balance field

Assignable

The default assignable clause is

```
//@ assignable \everything;
```

Pure methods are

```
//@ assignable \nothing;
```

Pure constructors are

```
//@ assignable this.*;
```

Multiple Clauses

Similarly for `ensures, invariant`:

```
requires P;
requires Q;
```

is equivalent to:

```
requires P && Q;
```

Exceptions / signals

```
/*@ requires balance < amount;
@   signals (Exception) balance == \old(balance);
@   signals_only Exception;
@ also
@   requires balance >= amount;
@   assignable balance;
@   ensures balance == \old(balance) - amount;
@*/
     public void debit(int amount) throws Exception {
           if (this.balance < amount) {
               throw new Exception("Operation not supported");
           } else {
               this.balance -= amount;
           }
      }
```

If the precondition does not hold, the method should throw exceptions.

signals specifies an exception and postconditions which should be satisfied when an exception occurs.

signals_only specifies an exception when the given precondition does not hold.

*The keyword **also** indicates that the method inherits specifications from superclasses.

DbC as Documentation

It provides **good documentation** (classes and interfaces)

> For each method of a class or interface, the contract says what it requires, if anything, from the client and what it ensures to the client

> Better for documentation than just code
>> **Even better than informal documentation** (code comments or Javadoc)

> **More abstract than code** (does not implement the algorithm, focus only on what is assumed and what must be achieved)

> Formal specification can be checked, comments cannot

> Increases the chances of being kept up-to-date with respect to the code (**informal documentation is often outdated**)

JML and OpenJML

JML annotations can be attached to Java programs by writing them directly into the Java source code files

JML annotations are ignored by JAVA (& JAVAC)
JML needs a tool to verify JML statements: OpenJML

“OpenJML is a program verification tool for Java programs that allows you to check the specifications of programs annotated in JML”

Download & Installation:
https://www.openjml.org/documentation/installation.shtml

Note that OpenJML currently supports Java 8 (not 7 or 9)

Example code – GitLab/Repo

Australian National University

Think about…

Design by Contract **records details of methods responsibilities**. It can be used for internally called methods to avoid constantly rechecking the validity of arguments. It can also be **used to specify the effect of calling a method**, and thus for **recording details in a design**. [1]

Which properties to specify?

Which level of detail?

Which tool should I use?

How expensive it can be? How to make it cost-effective?

Which kind of projects should I use JML?

Is it the best way to document software?

Exercises

Which alternative is INCORRECT about design by contract:

a) Interfaces form a contract between the class and the outside world, and this contract is enforced at runtime.
b) Design by contract can be an effective way for debugging, testing and, more generally, quality assurance. Additionally, it is a method for documenting software components.
c) The basic premise of design by contract is to associate a specification with every software element. These specifications, also known as contracts, govern the interaction of the element with the rest of the world.
d) Preconditions specify conditions that must hold before a method completes.

Exercises

Which one of the following alternatives is INCORRECT about JML?

a) \old(balance) is used to refer to the value of the variable balance at the time of entry into a method.

b) Precondition for a method can be specified using the instruction “requires”.

c) In the JML expression “//@ requires amount <= getBalance();” , the method getBalance() is not a pure method.

d) Postcondition for a method can be specified using the instruction “ensures”.

Meme for today's lecture! Keep practicing!

IN ORDER TO AVOID SHODDY MISTAKES, EVERYTHING WE DO FROM NOW ON WILL BE PART OF A DOCUMENTED PROCESS.
WHAT DOCUMENTED PROCESS DID YOU USE TO DECIDE WHAT DOCUMENTED PROCESS TO USE?
OR IS THIS ONE OF THOSE SHODDY MISTAKES I KEEP HEARING ABOUT?
www.dilbert.com

© Scott Adams, Inc./Dist. by UFS, Inc.

"My code was my documentation", said a *former* developer.

Recommended Reading

JML reference manual

http://www.eecs.ucf.edu/~leavens/JML/refman/jmlrefman.pdf

General information

http://www.eecs.ucf.edu/~leavens/JML//index.shtml