CS357 Software Verification

Design by Contract

Lecturer:Rosemary Monahan

21 September 2017

Design by Contract: Outline

- Introduction
- Contracts
- Class Invariants

Design by Contract: Outline

- Introduction
- Contracts
- Class Invariants

Design by Contract

- Popularised/Advocated by Bertrand Meyer
- Fundamentally connected with OO design
- Originally implemented in the Eiffel programming language
- Lately manifested in C++, Java, C# etc. - still an active research area
- “Design by Contract” is a trademark - the generic name is *Programming by Contract*

DBC: Key concept

Every class has two “roles” associated with it:

1. The supplier
   Wrote the class code; documents it, maintains it; knows about the class *implementation*; publishes the class *interface*
2. The client
   Uses the class in their own code; reads the documentation (presumably); knows about the class *interface*; knows nothing about the implementation

Possible pitfalls with the “conventional” approach

Without DbC:

- The client may not understand in what situations a method can be used
- The client may not understand fully the consequences of running a method
- When the supplier changes the implementation, how do they ensure the interface is preserved? (big rule: do not break your clients’ old code)
- If the interface does change, how does the supplier ensure *some* consistency?

Design by Contract: Main Principle

Main principles of DbC:

- Every public method has a precondition and a postcondition
- The precondition expresses the constraints under which the method will function properly
- The postcondition expresses what will happen when a method executes properly

Preconditions and postconditions are boolean-valued expressions - i.e. they evaluate to *true*/*false*

Design by Contract: Outline

- Introduction
- Contracts
- Class Invariants

The Contract

Defining preconditions and postconditions establishes a contract from the supplier to the client:

- IF the *client* runs the method in situations that satisfy the precondition
- THEN the *supplier* will make sure that the method execution will always deliver a state that satisfies the postcondition

Contract: Obligations and Benefits

	Precondition	Postcondition
Client	*obligation*: I must ensure the precondition is true before calling the method	*benefit*: I'm guaranteed the postcondition will be established by the method
Supplier	*benefit*: I can assume the precondition is true when writing the method	*obligation*: I must ensure when I write the method that the postcondition is satisfied

Preconditions: the supplier's benefit

Non-redundancy principle: The body of a method shall not check to see if the precondition is true

Preconditions: the supplier's benefit

Non-redundancy principle: The body of a method shall not check to see if the precondition is true

- This DBC principle is the opposite of *defensive programming*
- DBC reduces overhead and complexity of those extra checks
- Also: maybe the client has already checked the precondition (for some other purpose) - how many times do we have to check it?

Precondition Availability

Precondition Availability Rule: Every feature appearing in the precondition of a routine must be available to every client to which the routine is available

Precondition Availability

Precondition Availability Rule: Every feature appearing in the precondition of a routine must be available to every client to which the routine is available

- This is a reasonable demand, since the *client* must establish the precondition before calling a method.
- So, for example, a method precondition should not refer to the private attributes of the class.
- Checking that this rule holds can be done by the compiler.
- The same is not true for the postcondition - it may refer to private attributes (Why?)

Important: Violating an Assertion

What happens if an assertion (pre- or post-condition) is *not* satisfied?

Important: Violating an Assertion

What happens if an assertion (pre- or post-condition) is *not* satisfied?

- Then when the code is run, an *assertion is violated*. This means there is a bug in the code.
- An *exception* should be thrown, telling the user what assertion has been violated (i.e. where it is), and how it has been violated.
- If a *precondition* is violated, then the bug is in the client's code
- If a *postcondition* is violated, then the bug is in the supplier's code

What assertions aren't

Assertions are not an input checking mechanism

- Assertions are software-to-software contracts.
- They are not software-to-user checks, or software-to-device checks.
- Erroneous input from devices/users should be dealt with by writing the appropriate code.

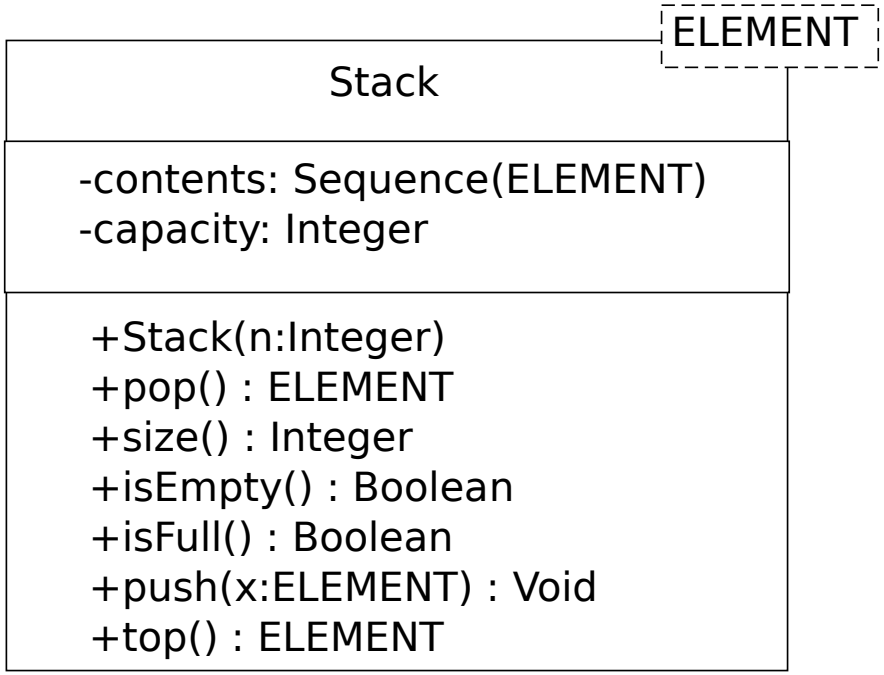
What assertions aren't

Assertions are not control structures

- A properly-working, bug-free piece of software should not throw assertions.
- Your code not try to catch an assertion-exception and fix things.
- (compare: *exceptions* are not a control structure)

Example: A Stack class in UML

Suppose we have a `Stack` class where the stack elements are of type `ELEMENT`



What are the pre- and post-conditions for the methods?

Design by Contract: Outline

- Introduction
- Contracts
- Class Invariants

Class Invariants

A class invariant is just an assertion involving the (class and) instance variables

- All objects of the class must satisfy the invariant at all stable times, i.e.:
  - After the execution of any of the constructors
  - Before and after the execution of every public method
- Corollary (1): Private methods *can* violate the invariant
- Corollary (2): It is the class *supplier*'s job to make sure the constructors and public methods maintain the invariant

Consequences of the class invariant

- If you don't define a constructor, then the default constructor (or the class initialisation code) must establish the invariant
- The class invariant is effectively *and*-ed with both the precondition and the postcondition of each public method.
- Consequences for the supplier when writing a public method:
  - *Benefit:* can assume that the invariant holds at the start of the method
  - *Obligation:* must ensure that the invariant holds at the end of the method

DBC and Inheritance

- How does DBC work with inheritance?
- What is the relationship between the class invariant of a superclass and that of a subclass?
- What is the relationship between method pre/post conditions in a superclass and those of an overriding version in a subclass?

Reminder: Inheritance

Assume *A* is a superclass of *B*; then

- *B* inherits all the (non-private) methods and instance variables from *A*.
- *B* might add more of these (*extending* the class), or change the implementation of some methods (*overriding* the method)
- If we have allowed for an *A*-object in the program code (statically), then at run-time we may use an A-object, or an object from any of *A*'s subclasses
- Therefore, a *B*-object must be able to do anything an *A*-object can do; i.e. it must be capable of behaving exactly like an *A*-object

(Liskov Substitution Principle)

Inheritance and overriding: example

```
// Supplier's code

class Courier
{
  public void deliver(Package p, Destination d)
  {
    // pre: Weight of package is less than 5kg
    // post: package is delivered within 3 working days
  }
}
```

What kind of precondition and postcondition should the `deliver` method in a subclass of `Courier` have?

Inheritance and overriding: example

```
// Client's code

public void sendByCourier(Courier c)
{
  Package p = new Package(...);
  Desintation d = new Desintation(...);
  .....
  // What must the client's code establish here?
  c.deliver(p,d);
  // What can the client's code assume here?
  .....
}
```

Method *pre*conditions and Inheritance

When *A* is a superclass of *B*...

- Remember: the *client* must satisfy the precondition
- ... but the client doesn't know the run-time type of the object
- ... so the client only knows about the preconditions of *A*-methods
- *Therefore:* Preconditions of methods in *B* cannot demand more than preconditions of methods in *A*. (They may demand less)

In logic terms, an *A*-method's precondition must *imply* the precondition of any overridden version of that method in *B*

Method *post*conditions and Inheritance

When *A* is a superclass of *B*...

- The supplier must ensure that every method establishes its postcondition
- Thus, every *B*-method that overrides an *A* method must (at least) establish the *A*-method's postcondition
- It may establish more than this (e.g. properties relating to new variables in the class)

In logic terms, a *B*-method's postcondition must *imply* the postcondition of any overridden version of that method in *A*

Inheritance and class invariants: example

```
// Supplier's code

class Courier
{
  class-invariant: insurance cover > € 1,000,000

  public void deliver(p : Package, d:Destination)
     .....

  .....

}
```

What kind of class invariant should a subclass of `Courier` have?

Class invariants and Inheritance

If *A* is a superclass of *B*:

- *B* inherits all the (non-private) class invariants from *A*
- *B* may add a few of its own, either on:
  - the inherited variables,
  - or on new variables introduced by *B*

In logic terms, *B*'s class invariants must *imply A*'s class invariants.

Summary: DBC and Inheritance

In summary, if *A* is a superclass of *B*, then you're writing *B*, you can:

- strengthen the class invariant
- weaken the *pre*condition of overridden methods
- strengthen the *post*condition of overridden methods

But not the opposite!!!

Further reading...

Book: *Object-Oriented Software Construction* (Second Edition) by Bertrand Meyer, Prentice-Hall, 1997.

Chapter 11: *Design by Contract: building reliable software*

Video: Design by Contract(TM) - Built in mechanism for bug prevention, http://www.eiffel.com/developers/presentations/