Software Correctness

- When is a class correct?
  - It's a relative concept; what is required?
  - But it's the correct question: the class is the basic independent, reusable unit of software
- Theory flashback: class = Abstract Data Type
  - Commands (*push*, *pop*, *empty*, *full*)
  - Axioms (*count == 0* iff *empty*)
  - Preconditions (*pop* requires *not empty*)
- Why isn't this reflected in programming?

Design by Contract

David Talby

Design by Contract

- Created by Bertrand Meyer, in Eiffel
- Each class defines a contract, by placing assertions inside the code
- Assertions are just Boolean expressions
  - Eiffel: identified by language keywords
  - iContract: identified by javadoc attributes
- Assertions have no effect on execution
- Assertions can be checked or ignored

Approaches to Correctness

- Testing
  - Tests only cover specific cases
  - Tests don't affect extensions (inheritance)
  - If something doesn't work, where is the problem?
  - It is difficult to (unit-) test individual classes
- Formal Verification
  - Requires math & logic background
  - Successful in hardware, not in software
- The *assert()* macro
  - Introduced to Java only in JDK 1.4

Methods II

- The same in iContract syntax:

```
//** return Square root of x
      @pre x >= 0
      @post return * return == x */
 double sqrt (double x) { … }
```

- Assertions are just Boolean expressions
  - Except *result* and *old* in postconditions
  - Function calls are allowed, but…
  - Don't modify data: *++i, inc(x), a = b*

Methods

- Each feature is equipped with a precondition and a postcondition

```
double sqrt (double x)
    require
          x >= 0
    do
          …
    ensure
          result * result == x
    end
```

Class Invariants

- Each class has an explicit invariant

```
class Stack[G]
private
    int count;
    boolean isEmpty() { … }
    … other things …
invariant
    isEmpty() == (count == 0)
end
```

The Contract

	Client (caller)	Supplier (feature)
Obligations:	fulfill precondition	fulfill postcondition
Benefits:	can assume postcondition	can assume precondition

When is a Class Correct?

- For every constructor:

  { Pre } code { Post ∧ Inv }

- For every public method call:

  { Pre ∧ Inv } code { Post ∧ Inv }

- Origin is Abstract Data Type theory
- Private methods are not in the contract
- *Undecidable* at compile time

Theory: Hoare Clauses

- Hoare's Notation for discussing correctness:

  { P } code { Q }

- For example:

  { x >= 10 } x = x + 2 { x >= 12 }

- Partial Correctness: If a program starts from a state satisfying P, runs the code and completes, then Q will be true.
- Full Correctness: If a program start from a state satisfying Q and runs the code, then eventually it will complete with Q being true.

Common Mistakes II

- Don't use defensive programming
  - The body of a routine must never check its pre- or post-conditions.
  - This is inefficient, and raises complexity.
- Don't hide the contract from clients
  - All the queries in a method's precondition must be at least as exported as the method
  - Doesn't have to be so in postconditions

Common Mistakes

- Not an input-checking mechanism
  - Use *if* to test human or machine output
  - Assertions are **always** true
- Not a control structure
  - Assertion monitoring can be turned off
  - They are applicative, not imperative, and must not include any side effects
  - Besides, exceptions are inefficient
- An assertion violation is always a bug
  - In precondition: client bug
  - In postcondition or invariant: supplier bug

Inheritance and DbC II

```
class Parent {
  void f() {
    require PPre
    ensure PPost
    …
  }
  invariant PInv
}
```

```
class Child extends Parent{
  void f() {
    require CPre
    ensure CPost
    …
  }
  invariant CInv
}
```

- Derivation is only legal if:
  - PPre → CPre
  - CPost → PPost
  - CInv → PInv

Inheritance and DbC

- The LSP Principle
  - Functions that use references to base classes must also work with objects of derived classes without knowing it.
  - Or: Derived classes inherit obligations as well
- How to break it
  - Derived method has a stronger precondition
  - Derived method has a weaker postcondition
  - Derived class does not obey parent's invariant

Loop Correctness

- Loops are hard to get right
  - Off-by-one errors
  - Bad handling of borderline cases
  - Failure to terminate
- There are two kinds of loops
  - Approximation (*while* and recursion)
  - Traversal (traditional *for*)

Inheritance and DbC III

- The Eiffel way
  - Child method's precondition is PPre ∨ CPre
  - Child method's postcondition is PPost ∧ CPost
  - Child's invariant is PInv ∧ CInv
  - This is how the runtime monitors assertions
- Abstract Specifications
  - Interfaces and Abstract methods can define preconditions, postconditions and invariants
  - A very powerful technique for frameworks

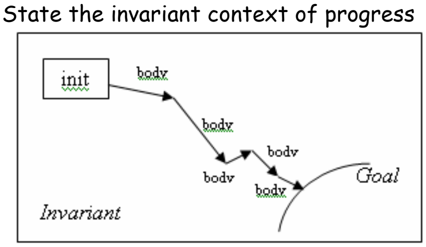
Approximation Loops II

- The loop is correct if:
  - Variant is a decreasing positive integer
  - Invariant is true before each iteration

```
int gcd(int a, int b) {
  int x = a, y = b;
  while (x != y)
    variant max(x, y)
    invariant x > 0 && y > 0 // && gcd(x,y)=gcd(a,b)
    do if (x > y) x = x - y; else y = y - x;
  return x;
}
```

Approximation Loops

- Prove that progress is made each step
- State the invariant context of progress



Why use Design by Contract?

- Speed – find bugs faster
- Testing – per class, including privates
- Reliability – runtime monitoring
- Documentation – part of the interface
- Reusability – see Ariane 5 crash
- Improving programming languages
  - Finding more bugs at compile time
  - Removing redundant language features

Traversal Loops

- Traverse a known collection or sequence
  - for (int i=0; i < 10; i++)
  - for (iterator<x> i = xlist.iterator(); ...)
- Invariant: Total number of elements
- Variant: Number of elements left
- Estimator: Number of elements left
- Can be imitated by approximation loops
  - Use *for* only when variant = estimator

The Missing Ingredient

Sometimes no checks should be done:

- A method's caller must ensure *x != null*
- *x* is never *null* "by nature"

> We must be able to state that ensuring a property is someone else's **responsibility**

- We must document it as well

An Example: Null Pointers

The #1 Java runtime error: NullPointerException
How do we know that a call's target is not null?

{? x != null} x.use {use postconditions}

- Out of context:
  x := new C; x.use;
- Because we checked:
  if (x != null) x.use;
  while (x != null) { x.use; foo(x); }
- But this is not enough!

Letting the Compiler Check II

- ADT Assertions:
  - precondition when feature begins
  - postcondition of called feature
  - the class invariant
- Incremental, per-feature check
- Test can be optional per class
- All compile-time, yet fully flexible

Letting the Compiler Check

- Rule: *x.use* does not compile if *x != null* can't can't be proved right before it
- Computation Assertions:
  - *x = new C*
  - *x = y*, assuming *y != null*
  - *if (x != null) …*
  - *while (x != null) ...*

The Big Picture

- Contracts complement what is learnt from code
- Identifying a simple kind of assertions is enough
  - But syntax is strict: *not (x == null)* won't work
- This works even though:
  - Assertions aren't trusted to be correct
  - They have no runtime cost, unless requested
- The same principle is used for language features
  - *x.foo(); y.foo();* can run in parallel iff *x != y*
  - *x.foo()* can bind statically if *x exact_instanceof C*

Sample Caught Bugs

- Infinite recursion:

  int count() { return 1 + left.count() + right.count(); }
- Forgotten initialization:

  Socket s = new BufferedSocket();
  s.getBuffer().write("x"); // s.connect() not yet called
- Neglecting the empty collection:

  do tok.getToken().print() while (!tok.done());
- Using uncertain results:

  f = filemgr.find(filename); f.delete();

DbC in Real Life: UML

- UML supports pre- and post-conditions as part of each method's properties
- Invariants are supported at class level
- Object Constraint Language is used
  - Formal language - not code
  - Readable, compared to its competitors
  - Supports *forall* and *exists* conditions

DbC in Real Life: C/C++

- In C, the assert macro expands to an if statement and calls abort if it's false

  assert(strlen(filename) > 0);
- Assertion checking can be turned off:

  #define NDEBUG
- In C++, redefine Assert to throw instead of terminating the program
- Every class should have an invariant
- Never use *if()* when *assert()* is required

Exceptions

- Definition: a method *succeeds* if it terminates in a state satisfying its contract. It *fails* if it does not succeed.
- Definition: An *exception* is a runtime event that may cause a routine to fail.
- Exception cases
  - An assertion violation (pre-, post-, invariant, loop)
  - A hardware or operating system problem
  - Intentional call to *throw*
  - A failure in a method causes an exception in its caller

DbC in Real Life: Java

- Assertions that can be turned on and off are only supported from JDK 1.4

  assert interval > 0 && interval <= 1 : interval;
- The most popular tool is iContract
  - Assertions are Javadoc-style comments
  - Instruments source code, handles inheritance
- Based on the OCL
  - *@invariant forall IEmployee e in getEmployees() | getRooms().contains(e.getOffice())*
  - *@post exists IRoom r in getRooms() | r.isAvailable()*

Improper Flow of Control

- Mistake 3: Using exceptions for control flow
  ```
  try { value = hashtable.find(key); }
  catch ( NotFoundException e ) { value = null; }
  ```
- It's bad design
  - The contract should never include exceptions
- It's extremely inefficient
  - Global per-class data is initialized and stored
  - Each *try*, *catch*, or exception specification cost time
  - Throwing an exception is *orders of magnitude slower* than returning from a function call

Disciplined Exception Handling

- Mistake 1: Handler doesn't restore stable state
- Mistake 2: Handler silently fails its own contract
- There are two correct approaches
  - Resumption: Change conditions, and retry method
  - Termination: Clean up and fail (re-throw exception)
- Correctness of a *catch* clause
  - Resumption: { True } Catch { Inv $\wedge$ Pre }
  - Termination: { True } Catch { Inv }

Goals

- Exception Neutrality
  - Exceptions raised from inner code (called functions or class T) are propagated well
- Weak Exception Safety
  - Exceptions (either from class itself or from inner code) do not cause resource leaks
- Strong Exception Safety
  - If a method terminates due to an exception, the object's state remains unchanged

Case Study: Genericity

- It's *very* difficult to write generic, reusable classes that handle exceptions well
  - Genericity requires considering exceptions from the template parameters as well
  - Both default and copy constructors may throw
  - Assignment and equality operators may throw
  - In Java: constructors, *equals()* and *clone()* may throw
- "A False Sense of Security"
  - Tom Cargill paper's on code for class Stack<T>
  - Affected design of STL, as well as Java containers
  - Among the conclusions: Exceptions affect class design

Summary

- Software Correctness & Fault Tolerance
- Design by Contract
  - When is a class correct?
  - Speed, Testing, Reliability, Documentation, Reusability, Improving Prog. Languages
- Exceptions
  - What happens when the contract is broken?
  - Neutrality, Weak Safety, Strong Safety