Introduction to contract programming

Łukasz Ziobroń

Nokia

Code Dive Community, 2015

Agenda

1. Problematic examples
2. DbC Theory
3. Language support
4. Own C++ DbC implementation
5. Summary

Section 1

Problematic examples

Ariane 5 mission

In 1996, the Ariane 5 rocket was reusing software from the Ariane 4. 37 seconds after its maiden launch the self-destruct safety mechanism was activated.

A data conversion from 64-bit floating point value to 16-bit signed integer value to be stored in a variable representing horizontal bias caused a processor trap because the floating point value was too large to be represented by a 16-bit signed integer. [1]

This bug, existing in Ariane 4 software never came out on Ariane 4 rocket.

[1] https://en.wikipedia.org/wiki/Ariane_5#Notable_launches

Ariane 5 mission

Basic calculator

```
1  int calculate(int a, int b, Operation op)
2  {
3      switch(op)
4      {
5          case Add:      return a + b;
6          case Subtract: return a - b;
7          case Multiply: return a * b;
8          case Divide:   return a / b;
9          default:       return 0; // should never get here
10     }
11 }
```

Basic calculator

```
 1  int calculateWithChecks(int a, int b, Operation op)
 2  {
 3      if(a < 0 || b < 0)
 4      {
 5          throw out_of_range("Number cannot be negative");
 6      }
 7
 8      switch(op)
 9      {
10          case Add:       return a + b;
11          case Subtract:  return a - b;
12          case Multiply:  return a * b;
13          case Divide:    if(b != 0.0)
14                          {
15                              return a / b;
16                          }
17                          else
18                          {
19                              throw logic_error("Division by zero");
20                          }
21          default:        throw logic_error("Undefined operation");
22      }
23  }
```

Basic calculator

```
 1  int calculateWithAssertions(int a, int b, Operation op)
 2  {
 3      assert(a >= 0 && b >= 0);
 4      switch(op)
 5      {
 6          case Add:      return a + b;
 7          case Subtract: return a - b;
 8          case Multiply: return a * b;
 9          case Divide:   assert(b != 0.0);
10                         return a / b;
11          default:       assert(false && "Undefined operation");
12      }
13  }
```

Basic calculator

```
1  int calculateWithContract(int a, int b, Operation op)
2  {
3      precondition(a >= 0 && b >= 0);
4      switch(op)
5      {
6          case Add:      return a + b;
7          case Subtract: return a - b;
8          case Multiply: return a * b;
9          default:       precondition(b != 0.0);
10                        return a / b;
11     }
12 }
```

Basic calculator - summary

What should we do when input is not valid?

Basic calculator - summary

What should we do when input is not valid?

- nothing (very clean code, but not recommended)

Basic calculator - summary

What should we do when input is not valid?

- nothing (very clean code, but not recommended)
- signal the problem (return optional, special or magic values - NaN, 0, none)

Basic calculator - summary

What should we do when input is not valid?

- nothing (very clean code, but not recommended)
- signal the problem (return optional, special or magic values - NaN, 0, none)
- throw an exception (and force the user to catch it)

Basic calculator - summary

What should we do when input is not valid?

- nothing (very clean code, but not recommended)
- signal the problem (return optional, special or magic values - NaN, 0, none)
- throw an exception (and force the user to catch it)
- terminate the application that is about to enter an UB

Basic calculator - summary

What should we do when input is not valid?

- nothing (very clean code, but not recommended)
- signal the problem (return optional, special or magic values - NaN, 0, none)
- throw an exception (and force the user to catch it)
- terminate the application that is about to enter an UB

Basic calculator - summary

What should we do when input is not valid?

- nothing (very clean code, but not recommended)
- signal the problem (return optional, special or magic values - NaN, 0, none)
- throw an exception (and force the user to catch it)
- terminate the application that is about to enter an UB

Proper behavior will be considered later

Blender interface

Acceptance Criteria

- blender cannot run empty
- speed can be changed only by 1
- speed range is from 0 to 9

Blender interface

Acceptance Criteria

- blender cannot run empty
- speed can be changed only by 1
- speed range is from 0 to 9

ATDD - Acceptance Test Driven Development

Blender interface

```
1   /**
2    * @invariant getSpeed() > 0 implies isFull()     // don't run empty blender
3    * @invariant getSpeed() >= 0 && getSpeed() < 10 // verify range
4    */
5   public interface Blender
6   {
7       int getSpeed();
8       boolean isFull();
9
10      /**
11       * @pre Math.abs(getSpeed() - x) <= 1 // change only by 1
12       * @pre x >= 0 && x < 10              // verify range
13       * @post getSpeed() == x
14       */
15      void setSpeed(final int x);
16
17      /**
18       * @pre !isFull()
19       * @post isFull()
20       */
21      void fill();
22
23      /**
24       * @pre isFull()
25       * @post !isFull()
26       */
27      void empty();
28  }
```

DB query example by A. Krzemieński

```
1  bool checkIfUserExists(std::string userName)
2  {
3    std::string query = "select␣count(*)␣from␣USERS␣where␣NAME␣=␣\'"
4                      + userName + "\';"
5    return DB::run_sql<int>(query) > 0;
6  }
7
8  bool autheticate()
9  {
10   std::string userName = UI::readUserInput();
11   return checkIfUserExists(userName);
12 }
```

Possible inputs

When end-user is nice...

Tom

Possible inputs

When end-user is nice...

Tom

When end-user is malicious...

JOHN'; delete from USERS where 'a' = 'a

Possible inputs

When end-user is nice...

Tom

When end-user is malicious...

JOHN'; delete from USERS where 'a' = 'a

Achtung!

SQL Injection vulnerability!

select count(*) from USERS where NAME = 'JOHN'; delete from USERS where 'a' = 'a';

Who is guilty?

```
1  bool checkIfUserExists(std::string userName)
2  {
3    std::string query = "select␣count(*)␣from␣USERS␣where␣NAME␣=␣\'"
4                        + userName + "\';"
5    return DB::run_sql<int>(query) > 0;
6  }
7
8  bool autheticate()
9  {
10   std::string userName = UI::readUserInput();
11   return checkIfUserExists(userName);
12 }
```

Assuming that functions *checkIfUserExists* and *authenticate* were written by two different people, which of them is responsible for the bug?

Who is guilty?

```
1  bool checkIfUserExists(std::string userName)
2  {
3    std::string query = "select␣count(*)␣from␣USERS␣where␣NAME␣=␣\'"
4                        + userName + "\';"
5    return DB::run_sql<int>(query) > 0;
6  }
7
8  bool autheticate()
9  {
10   std::string userName = UI::readUserInput();
11   return checkIfUserExists(userName);
12 }
```

Assuming that functions *checkIfUserExists* and *authenticate* were written by two different people, which of them is responsible for the bug?

Without clearly stated expectations, it is impossible to tell whose fault it is: it is a failure to communicate between two programmers (or even one).

Let's fix it!

We have two problems then:

- the program has a bug
- it is not clear whose responsibility it is

Let's fix it!

We have two problems then:

- the program has a bug
- it is not clear whose responsibility it is

Implementation of name validation

```
bool isValidName( std::string const& text )
{
  const static std::regex NAME{"\\w+"};
  std::smatch match;
  return std::regex_match(text, match, NAME);
}
```

Let's fix it!

```
1  // NOT RECOMMENDED!
2  bool checkIfUserExists(std::string userName)
3  {
4    if (!isValidName(userName)) SIGNAL();
5
6    std::string query = "select␣count(*)␣from␣USERS␣where␣NAME␣=␣\'"
7                        + userName + "\';"
8    return DB::run_sql<int>(query) > 0;
9  }
10
11 bool autheticate()
12 {
13   std::string userName = UI::readUserInput();
14   if (!isValidName(userName)) SIGNAL();
15
16   return checkIfUserExists(userName);
17 }
```

Let's fix it!

```
1  // NOT RECOMMENDED!
2  bool checkIfUserExists(std::string userName)
3  {
4    if (!isValidName(userName)) SIGNAL();
5
6    std::string query = "select␣count(*)␣from␣USERS␣where␣NAME␣=␣\'"
7                        + userName + "\';"
8    return DB::run_sql<int>(query) > 0;
9  }
10
11 bool autheticate()
12 {
13   std::string userName = UI::readUserInput();
14   if (!isValidName(userName)) SIGNAL();
15
16   return checkIfUserExists(userName);
17 }
```

Nobody trusts nobody and everyone just checks for the dangerous conditions.

Fixed!

Problem fixed, but anothers arised

Problems:

- Performance: checking the same condition many times

Problem fixed, but anothers arised

Problems:

- Performance: checking the same condition many times
- Readability: code becomes messy

Problem fixed, but anothers arised

Problems:

- Performance: checking the same condition many times
- Readability: code becomes messy
- Flow: if SIGNAL() throws someone needs to catch the exception

Problem fixed, but anothers arised

Problems:

- Performance: checking the same condition many times
- Readability: code becomes messy
- Flow: if SIGNAL() throws someone needs to catch the exception

Problem fixed, but anothers arised

Problems:

- Performance: checking the same condition many times
- Readability: code becomes messy
- Flow: if SIGNAL() throws someone needs to catch the exception

What should SIGNAL() do?

Problem fixed, but anothers arised

Problems:

- Performance: checking the same condition many times
- Readability: code becomes messy
- Flow: if SIGNAL() throws someone needs to catch the exception

What should SIGNAL() do?

Whatever we choose, the program will grow in complexity; and complexity (especially a messy one like this) is likely to cause bugs.

Section 2

DbC Theory

What is Design by Contract?

A paradigm which was first introduced by Bertrand Meyer, the creator of Eiffel. Although Eiffel has support for programming by contract built into the language, most of the concepts can be used in any language [1].

Basically programming by contract creates a contract between the software developer and software user - in Meyer's terms the supplier and the consumer.

[1] http://www.cs.unc.edu/ stotts/COMP204/contract.html

What is Design by Contract?

A paradigm which was first introduced by Bertrand Meyer, the creator of Eiffel. Although Eiffel has support for programming by contract built into the language, most of the concepts can be used in any language [1].

Basically programming by contract creates a contract between the software developer and software user - in Meyer's terms the supplier and the consumer.

[1] http://www.cs.unc.edu/ stotts/COMP204/contract.html

3 assumptions

Every feature, or method, starts with a **precondition** that must be satisfied by the consumer of the routine.

And each feature ends with **postconditions** which the supplier guarantees to be true (if and only if the preconditions were met).

Also, each class has an **invariant** which must be satisfied after any changes to the object represented by the class. In the other words, the invariant guarantees the object is in a valid state.

3 assumptions

Every feature, or method, starts with a **precondition** that must be satisfied by the consumer of the routine.

And each feature ends with **postconditions** which the supplier guarantees to be true (if and only if the preconditions were met).

Also, each class has an **invariant** which must be satisfied after any changes to the object represented by the class. In the other words, the invariant guarantees the object is in a valid state.

3 assumptions

Every feature, or method, starts with a **precondition** that must be satisfied by the consumer of the routine.

And each feature ends with **postconditions** which the supplier guarantees to be true (if and only if the preconditions were met).

Also, each class has an **invariant** which must be satisfied after any changes to the object represented by the class. In the other words, the invariant guarantees the object is in a valid state.

Precondition example

```
1  double sqrt( double r )
2  in // start a block with preconditions
3  {
4      r > 0.0: throw bad_input();
5  }
6  do // normal function body
7  {
8      ...
9  }
```

Postcondition example

```
1  int foo( int& i )
2  out // start block with postconditions
3  {
4       i == in i + 1;
5       return % 2 == 0;
6  }
7  do // normal function body
8  {
9       i++;
10      return 4;
11 }
```

Class invariant example

```
1  class container
2  {
3      // ...
4      invariant
5      {
6          size() >= 0;
7          size() <= max_size();
8          is_empty() implies size() == 0;
9          is_full() implies size() == max_size();
10         distance( begin(), end() ) == size();
11     }
12 };
```

Example from latest C++ DbC proposal

```
 1  int factorial( int n )
 2  precondition
 3  {
 4      0 <= n && n <= 12;
 5  }
 6  postcondition( result )
 7  {
 8      result >= 1;
 9  }
10  {
11      if ( n < 2 )
12          return 1;
13      else
14          return n * factorial( n - 1 );
15  }
```

Example from latest C++ DbC proposal

```
1  int factorial( int n )
2  precondition
3  {
4      0 <= n && n <= 12;
5  }
6  postcondition( result )
7  {
8      result >= 1;
9  }
10 {
11     if ( n < 2 )
12         return 1;
13     else
14         return n * factorial( n - 1 );
15 }
```

Examples from C++ Design by Contract proposal: n1962, n1613.

Section 3

Language support

DbC support in some languages

- C++
  - GNU Nana: https://savannah.gnu.org/projects/nana
  - Escher C++ Verifier: http://eschertech.com/products/ecv.php
  - Loki Contract Checker: http://loki-lib.sourceforge.net
  - Contract++: http://sourceforge.net/p/contractpp/wiki/Home
- Java
  - **iContract**/jContracts: http://sourceforge.net/projects/jcontracts
  - valid4j: https://github.com/helsing/valid4j
  - jContractor: http://jcontractor.sourceforge.net
- Python
  - PyContracts: https://pypi.python.org/pypi/PyContracts
  - PyDBC: https://pypi.python.org/pypi/PyDBC
- .Net
  - Code Contracts: http://research.microsoft.com/en-us/projects/contracts
- **D**: built-in mechanism
- **Eiffel**: built-in mechanism

Eiffel - it started here

```
1  put( x: ELEMENT; key: STRING ) is
2  require
3      count <= capacity
4      not key.empty
5  do
6      ... Some insertion algorithm ...
7  ensure
8      has( x )
9      item( key ) = x
10     count = old count + 1
11 end
```

Eiffel - it started here

```
1  class ACCOUNT
2  invariant
3      consistent_balance: balance = all_deposits.total
4      ... rest of class ...
5  end
```

D - built-in mechanism

```
 1  int add_one( int i )
 2  in
 3  {
 4      assert( i > 0 );
 5  }
 6  out( result )
 7  {
 8      assert( result == i + 1 );
 9  }
10  body
11  {
12      return i + 1;
13  }
```

D - built-in mechanism

```
 1  class Date
 2  {
 3      int day;
 4      int hour;
 5      invariant
 6      {
 7          assert( 1 <= day && day <= 31 );
 8          assert( 0 <= hour && hour < 24 );
 9      }
10  }
```

iContract - quick overview

```
1  /**
2   * @pre f >= 0.0
3   * @post Math.abs((return * return) - f) < 0.001
4   */
5  public float sqrt(float f)
6  {
7      ...
8  }
9
10 /**
11  *  Append an element to a collection.
12  *
13  *  @post c.size() == c@pre.size() + 1
14  *  @post c.contains(o)
15  */
16 public void append(Collection c, Object o)
17 {
18     ...
19 }
```

Java - iContract: class invariants

```
1  /**
2   *  A PositiveInteger is an Integer that is guaranteed to be posit
3   *
4   *  @inv intValue() > 0
5   */
6  class PositiveInteger extends Integer
7  {
8      ...
9  }
```

Java - iContract: forall & exists quantifiers

```
1  /**
2   * @invariant forall IEmployee e in getEmployees() |
3   *                getRooms().contains(e.getOffice())
4   */
```

```
1  /**
2   * @post exists IRoom r in getRooms() | r.isAvailable()
3   */
```

Java - iContract: implications and logical operators

```
1  /**
2   * @invariant getRooms().isEmpty() implies
3   *            getEmployees().isEmpty() // no rooms, no employees
4   */
```

```
1  /**
2   * @invariant forall IEmployee e1 in getEmployees() |
3   *            forall IEmployee e2 in getEmployees() |
4   *            (e1 != e2) implies e1.getOffice() != e2.getOffice()
5   *            // a single office per employee
6   */
```

Java - iContract: Stack interface example

```
1  /**
2   *  @inv !isEmpty() implies top() != null
3   */
4  public interface Stack
5  {
6      /**
7       *  @pre o != null
8       *  @post !isEmpty()
9       *  @post top() == o
10      */
11     void push(Object o);
12     /**
13      *  @pre !isEmpty()
14      *  @post @return == top()@pre
15      */
16     Object pop();
17     /**
18      *  @pre !isEmpty()
19      */
20     Object top();
21     boolean isEmpty();
22 }
```

Java - iContract: Stack interface example

```
1  /**
2   *  @inv !isEmpty() implies top() != null
3   */
4  public interface Stack
5  {
6      /**
7       *  @pre o != null
8       *  @post !isEmpty()
9       *  @post top() == o
10      */
11     void push(Object o);
12     /**
13      *  @pre !isEmpty()
14      *  @post @return == top()@pre
15      */
16     Object pop();
17     /**
18      *  @pre !isEmpty()
19      */
20     Object top();
21     boolean isEmpty();
22 }
```

Hey man, aren't they a unit tests?

Java - iContract: Stack interface example

```
1  /**
2   *  @inv !isEmpty() implies top() != null
3   */
4  public interface Stack
5  {
6      /**
7       *  @pre o != null
8       *  @post !isEmpty()
9       *  @post top() == o
10      */
11     void push(Object o);
12     /**
13      *  @pre !isEmpty()
14      *  @post @return == top()@pre
15      */
16     Object pop();
17     /**
18      *  @pre !isEmpty()
19      */
20     Object top();
21     boolean isEmpty();
22 }
```

Hey man, aren't they a unit tests?

Maybe I can skip them?

Java - iContract: Stack interface example

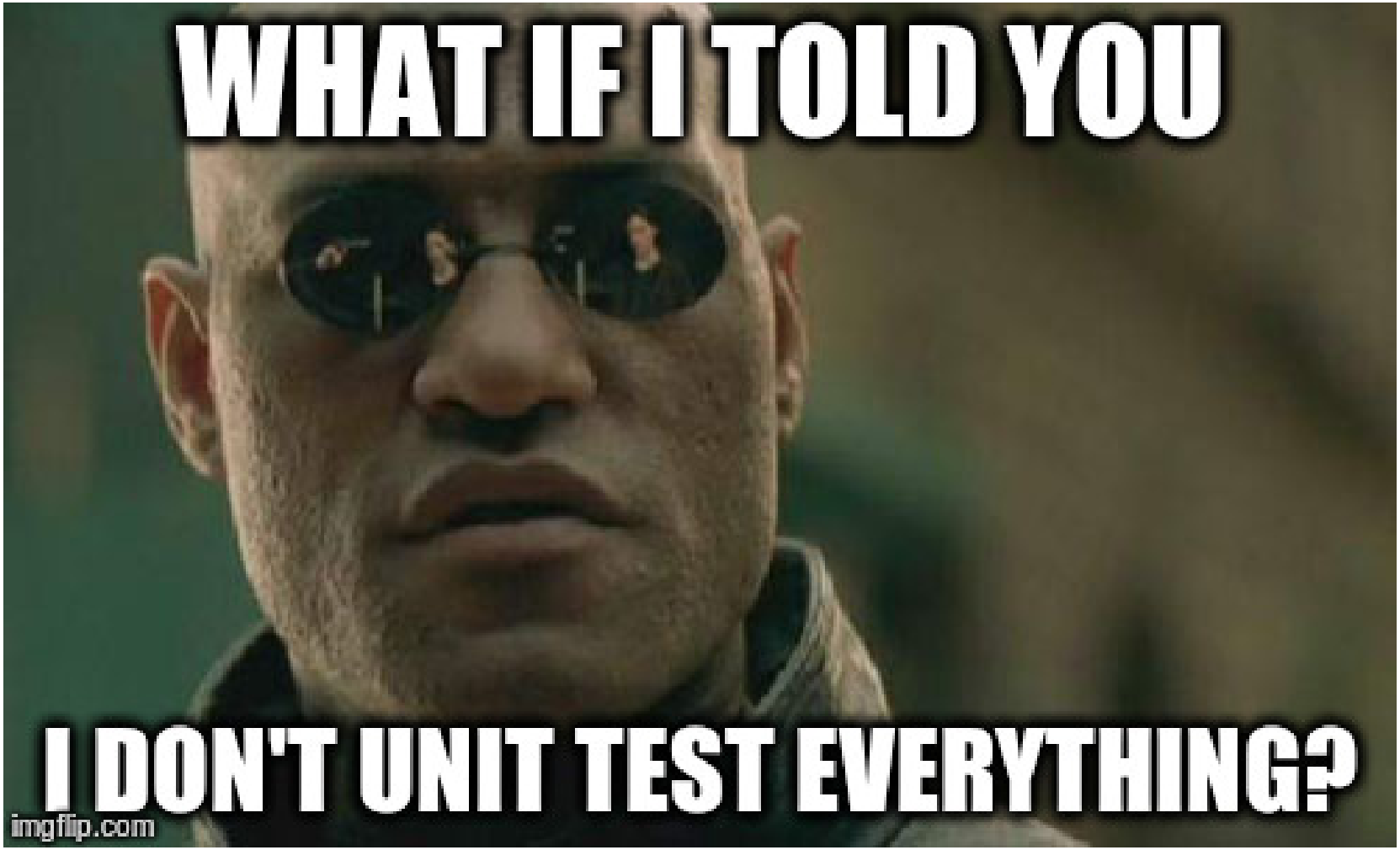
```
1  /**
2   *  @inv !isEmpty() implies top() != null
3   */
4  public interface Stack
5  {
6      /**
7       *  @pre o != null
8       *  @post !isEmpty()
9       *  @post top() == o
10      */
11     void push(Object o);
12     /**
13      *  @pre !isEmpty()
14      *  @post @return == top()@pre
15      */
16     Object pop();
17     /**
18      *  @pre !isEmpty()
19      */
20     Object top();
21     boolean isEmpty();
22 }
```

Hey man, aren't they a unit tests?

Maybe I can skip them?

You can, but don't forget to test your contracts.

Java - iContract: Stack interface example



Java - iContract: testing contract

```
1  /**
2   *  @inv isEmpty() implies elements.size() == 0
3   */
4  public class StackImpl implements Stack
5  {
6      private final LinkedList elements = new LinkedList();
7      public void push(Object o)
8      {
9          elements.add(o);
10     }
11     public Object pop()
12     {
13         final Object popped = top();
14         elements.removeLast();
15         return popped;
16     }
17     public Object top()
18     {
19         return elements.getLast();
20     }
21     public boolean isEmpty()
22     {
23         return elements.size() == 0;
24     }
25 }
```

Java - iContract: testing contract

```
1  public  class StackTest
2  {
3      public static void main(String[] args)
4      {
5          final Stack s = new StackImpl();
6          s.push("one");
7          s.pop();
8          s.push("two");
9          s.push("three");
10         s.pop();
11         s.pop();
12         s.pop();     //  causes an assertion to fail
13     }
14 }
```

Java - iContract: testing contract

```
1  public  class StackTest
2  {
3      public static void main(String[] args)
4      {
5          final Stack s = new StackImpl();
6          s.push("one");
7          s.pop();
8          s.push("two");
9          s.push("three");
10         s.pop();
11         s.pop();
12         s.pop();     //  causes an assertion to fail
13     }
14 }
```

```
1  $ java -cp ./instr StackTest
2  Exception in thread "main" java.lang.RuntimeException:
3  java.lang.RuntimeException: src/StackImpl.java:22: error:
4  precondition violated (StackImpl::top()): (/*declared in Stack::top()*/ (!isEmpty()))
5          at StackImpl.top(StackImpl.java:210)
6          at StackImpl.pop(StackImpl.java:124)
7          at StackTest.main(StackTest.java:15)
```

Java - iContract: Summary

How does it work?

- iContract preprocessor checks the JavaDoc comments for keywords

Java - iContract: Summary

How does it work?

- iContract preprocessor checks the JavaDoc comments for keywords
- iContract generates additional code for contract validation

Java - iContract: Summary

How does it work?

- iContract preprocessor checks the JavaDoc comments for keywords
- iContract generates additional code for contract validation
- iContract wraps function calls with generated precondition and postcondition checks

Java - iContract: Summary

How does it work?

- iContract preprocessor checks the JavaDoc comments for keywords
- iContract generates additional code for contract validation
- iContract wraps function calls with generated precondition and postcondition checks

Java - iContract: Summary

How does it work?

- iContract preprocessor checks the JavaDoc comments for keywords
- iContract generates additional code for contract validation
- iContract wraps function calls with generated precondition and postcondition checks

Summary

- JavaDoc style

Java - iContract: Summary

How does it work?

- iContract preprocessor checks the JavaDoc comments for keywords
- iContract generates additional code for contract validation
- iContract wraps function calls with generated precondition and postcondition checks

Summary

- JavaDoc style
- Contracts are in the documentation and in the code

Java - iContract: Summary

How does it work?

- iContract preprocessor checks the JavaDoc comments for keywords
- iContract generates additional code for contract validation
- iContract wraps function calls with generated precondition and postcondition checks

Summary

- JavaDoc style
- Contracts are in the documentation and in the code
- Users / callers can see your contract

Java - iContract: Summary

How does it work?

- iContract preprocessor checks the JavaDoc comments for keywords
- iContract generates additional code for contract validation
- iContract wraps function calls with generated precondition and postcondition checks

Summary

- JavaDoc style
- Contracts are in the documentation and in the code
- Users / callers can see your contract
- Less whitebox testing

Java - iContract: Summary

How does it work?

- iContract preprocessor checks the JavaDoc comments for keywords
- iContract generates additional code for contract validation
- iContract wraps function calls with generated precondition and postcondition checks

Summary

- JavaDoc style
- Contracts are in the documentation and in the code
- Users / callers can see your contract
- Less whitebox testing
- Testing contracts / interactions

Section 4

Own C++ DbC implementation

Let's implement the contract mechanism

```
1  #define preCondition(c)  assert(c)
2  #define postCondition(c) assert(c)
```

In mentioned proposals preconditions and postconditions are by default implemented with assertions, but they can be customized.

Let's implement the contract mechanism

```
1  #define preCondition(c)  assert(c)
2  #define postCondition(c) assert(c)
```

In mentioned proposals preconditions and postconditions are by default implemented with assertions, but they can be customized.

```
1  #define preCondition(c) \
2    if(!(c)) { \
3      std::cerr << __FILE__ << ":" << __LINE__ << ":␣" << #c; \
4      throw PreconditionException("preCondition␣" #c "␣failed!"); \
5    }
6  #define postCondition(c) \
7    if(!(c)) { \
8      std::cerr << __FILE__ << ":" << __LINE__ << ":␣" << #c; \
9      throw PostconditionException("postCondition␣" #c "␣failed!");
10   }
```

Let's implement the contract mechanism

Let's implement the contract mechanism

```
1  #ifdef NDEBUG
2  #define preCondition(c)  assert(c)
3  #define postCondition(c) assert(c)
4  #else
5  #define preCondition(c) \
6    if(!(c)) { \
7      std::cerr << __FILE__ << ":" << __LINE__ << ": " << #c; \
8      throw PreconditionException("preCondition " #c " failed!"); \
9    }
10 #define postCondition(c) \
11   if(!(c)) { \
12     std::cerr << __FILE__ << ":" << __LINE__ << ": " << #c; \
13     throw PostconditionException("postCondition " #c " failed!");
14   }
15 #endif
```

Custom DbC C++ function template

```
1  void SomeClass::someMethod(AnotherClass *fillMeWithData)
2  {
3      preCondition(fillMeWithData != nullptr);
4
5      // your code goes here...
6
7      postCondition(fillMeWithData->hasData());
8      postCondition(checkInvariant());
9  }
```

Custom DbC C++ function template

```
1  void SomeClass::someMethod(AnotherClass *fillMeWithData)
2  {
3      preCondition(fillMeWithData != nullptr);
4
5      // your code goes here...
6
7      postCondition(fillMeWithData->hasData());
8      postCondition(checkInvariant());
9  }
```

Any problems with this mechanism? (despite macros)

Custom DbC C++ function template

```
1  void SomeClass::someMethod(AnotherClass *fillMeWithData)
2  {
3      preCondition(fillMeWithData != nullptr);
4
5      // your code goes here...
6
7      postCondition(fillMeWithData->hasData());
8      postCondition(checkInvariant());
9  }
```

Any problems with this mechanism? (despite macros)

Propagation in inheritance. You lose your contractual agreement when you override someMethod in derived class.

Custom DbC C++ function template

```
1  void SomeClass::someMethod(AnotherClass *fillMeWithData)
2  {
3      preCondition(fillMeWithData != nullptr);
4
5      // your code goes here...
6
7      postCondition(fillMeWithData->hasData());
8      postCondition(checkInvariant());
9  }
```

Any problems with this mechanism? (despite macros)

Propagation in inheritance. You lose your contractual agreement when you override someMethod in derived class.

Solution: Non virtual interface (NVI idiom)

Non-virtual interface example

```
 1  class Base
 2  {
 3  public:
 4      void someMethod(AnotherClass * ptrData) // non-virtual
 5      {
 6          preCondition(ptrData != nullptr);
 7          someMethodImpl(ptrData);
 8          postCondition(ptrData->hasData());
 9          postCondition(checkInvariant());
10      }
11      virtual ~Base() {}
12  private:
13      virtual void someMethodImpl(AnotherClass * ptrData) = 0;
14      bool checkInvariant() { /*...*/ return true; }
15  };
16
17  class SomeClass : public Base
18  {
19  private:
20      void someMethodImpl(AnotherClass * ptrData) override
21      { /* your code goes here... */ }
22  };
```

Possible further code development

- pre and postconditions as callbacks (std::function, function ptrs, functors, lambdas)
- pre and postconditions as template parameters
- use constexpr functions as preconditions
- use switch to change behavior during runtime

Assert or throw?

ACHTUNG!

Calling a function whose precondition is not satisfied results in *undefined behavior*

Assert or throw?

ACHTUNG!

Calling a function whose precondition is not satisfied results in *undefined behavior*

The caller is not allowed to make any assumptions about the results of the function call.
It allows the function author to do the following things:

1. Optimize code based on assumption that precondition always holds.

Assert or throw?

ACHTUNG!

Calling a function whose precondition is not satisfied results in *undefined behavior*

The caller is not allowed to make any assumptions about the results of the function call.
It allows the function author to do the following things:

1. Optimize code based on assumption that precondition always holds.
2. Verify the precondition and report the violation by any means (call std::terminate, throw an exception, launch debugger).

Assert or throw?

ACHTUNG!

Calling a function whose precondition is not satisfied results in *undefined behavior*

The caller is not allowed to make any assumptions about the results of the function call.
It allows the function author to do the following things:

1. Optimize code based on assumption that precondition always holds.
2. Verify the precondition and report the violation by any means (call std::terminate, throw an exception, launch debugger).
3. Pick 1 or 2 based on other factors (like the value of macro NDEBUG).

Assert or throw?

The caller may assume nothing and cannot rely on current implementation, if he does not satisfy preconditions.

Section 5

Summary

Pros

- Eliminated redundant checks
- Less code
- Lower cyclomatic complexity
- Less time spent on adding new features
- Many bugs are detected ASAP
- Reduced time spent in debugger
- Safer code
- Protection against user / callers

Before you call the functionality ready, think about the contract - preconditions, postconditions and invariants.
The contract protects you against bugs and users as well. Users cannot blame you if they do not satisfy the preconditions.

Cons

- Lots of potential UB (nullptr, std::vector::operator[])
- Not all bugs can be detected
- Bugs in contracts (logical conditions)
- Difficulty to introduce contracts to big projects
- No native support in popular languages (C++, Java)
- Cannot evaluate conditions in compile time
- Performance penalty for checking contracts

Cons

You can try to disable contracts when development is finished and you know that client won't break anything.

Q & A

Questions?

Questions for audience

- Who have heard about contract programming earlier?

Questions for audience

- Who have heard about contract programming earlier?
- Who have used contract programming before?

Questions for audience

- Who have heard about contract programming earlier?
- Who have used contract programming before?
- Who thinks that contract programming is useful and helpful?

Questions for audience

- Who have heard about contract programming earlier?
- Who have used contract programming before?
- Who thinks that contract programming is useful and helpful?
- Who wants to hear more?

Resources


A. Hunt, D. Thomas
The Pragmatic Programmer. From journeyman to master.


A. Krzemieński
Preconditions - part I, part II, part III, part IV
https://akrzemi1.wordpress.com/2013/01/04/preconditions-part-i/
https://akrzemi1.wordpress.com/2013/02/11/preconditions-part-ii/
https://akrzemi1.wordpress.com/2013/03/13/preconditions-part-iii/
https://akrzemi1.wordpress.com/2013/04/18/preconditions-part-iv/


L. Crowl, T. Ottosen
Proposal to add Contract Programming to C++ (revision 4)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1962.html


D. Stotts
Programming by Contract
http://www.cs.unc.edu/ stotts/COMP204/contract.html


O. Enseling
iContract: Design by Contract in Java
http://www.javaworld.com/article/2074956/learn-java/icontract–design-by-contract-in-java.html


D. Brzeziński
Programowanie przez kontrakt
http://www.cs.put.poznan.pl/dbrzezinski/teaching/po/7 - Programowanie przez kontrakt.pdf

Contract programming advanced

- Some examples of silly bugs
- Analysis of precondition and postcondition code
- Implementation of more professional DbC framework in C++
- Comparison of performance between presented solutions
- Code bloat - how messy the code can be - comparison