Methodical Design by Contract Using Assertions and Exceptions in the Java™ Programming Language

Richard Mitchell
Vladimir Bacvanski
Petter Graff

InferData Corporation
From This Session You Will Learn...

A method for
Design by Contract,
and supporting practices
Learning Objectives

• As a result of this presentation, you will:

  – Understand six principles of writing contracts (topic 1)

  – Know two ways to exploit the relationship between exceptions and contracts (topic 2)
Speaker’s Qualifications

- Richard Mitchell trains and consults in object technology for InferData Corp

- Richard is lead author of the 2002 Addison Wesley book *Design by Contract, by Example*. He has used software contracts in programming and systems analysis for 20 years.
public interface MyCollection {

/**
 * Remove o from the collection
 * @require !( o == null )
 * @ensure !contains( o )
 */

public void remove( Object o );
public interface MyCollection {

/**
 * Remove o from the collection
 *
 * @require !( o == null )
 * @ensure !contains( o )
 */

public void remove( Object o );
public interface MyCollection {

    /**
     * Remove o from the collection
     * @require !( o == null )
     * @ensure !contains( o )
     */
    public void remove( Object o );

    Precondition disallows null argument.

    Postcondition clarifies that method removes ALL occurrences of o
    (time to improve the comment!)
The Parties to a Contract

The `remove()` method has a precondition and a postcondition.
The Parties to a Contract

I am writing a call to `remove(...)` so I must fulfill the precondition

The `remove()` method has a precondition and a postcondition

public void remove(...) precondition

postcondition
The Parties to a Contract

I am writing a call to `remove(...)` so I must fulfill the precondition.

I am implementing `remove(...)` so I must fulfill the postcondition.

The `remove()` method has a precondition and a postcondition.
The Parties to a Contract

I am writing a call to `remove(...)` so I must fulfill the precondition.

And I am the runtime support. Switch me on, and I’ll check they keep their promises!

I am implementing `remove(...)` so I must fulfill the postcondition.
Examples use the iContract tool, designed by Reto Kramer, and available at

www.reliable-systems.com

The tool parses tagged comments in your source file, and writes a new source file with code to invoke runtime checks of preconditions, post conditions and invariants

You could also use, for example, JContract from Parasoft
Key Benefits

Design by Contract yields these returns on investment:

✓ Strong support for debugging
✓ Explicit test oracles
✓ Trustworthy documentation

And it’s as easy as programming!
Agenda

Two Related Topics:

• TOPIC 1: A method for designing contracts
  – Example: Contracts for a queue interface
  – Demo: A false contract, a testing success

• TOPIC 2: Contracts and exceptions
  – Two different approaches
TOPIC 1

A Method for Designing Contracts

(Based on SIX Principles)
public interface Queue {

    public void add( Object o );
    public Object getAndRemove();
    public int size();
    public Object head();
    public boolean isEmpty();
}


PRINCIPLE 1: Separate Commands and Queries

Commands

```
public void add(Object o);
public void remove(); //just a command
```

Queries

```
public int size();
public Object head();
public boolean isEmpty();
```

We’ll bring back hybrid command/query later
PRINCIPLE 2: Separate Basic and Derived Queries

Basic queries

```java
public int size();
public Object head();
```

Derived queries

```java
public boolean isEmpty();
   -- we can derive isEmpty() from size()
```
PRINCIPLE 2: Separate Basic and Derived Queries

Basic queries

```java
public int size();
public Object head();
```

Derived queries

```java
public boolean isEmpty();
   -- we can derive isEmpty() from size()
```

Choice of basic queries can change as we develop contracts
/**
 * Is the queue empty?
 * @ensure return == ( size() == 0 )
 */

public boolean isEmpty();

-- if we know the value of size(),
-- we also know the value of
-- isEmpty()
PRINCIPLE 4: Specify Commands in Terms of Basic Queries

/**
 * Add o at the tail of the queue
 *  
 * @ensure size() == size()@pre + 1
 */

public void add( Object o );
PRINCIPLE 4: Specify Commands in Terms of Basic Queries

/**
 * Add o at the tail of the queue
 *
 * @ensure size() == size() + 1
 */

public void add( Object o );

This is not yet a good contract
E.g., where does the added object go?
More on Principle 4 later
/**
 * The first element in the queue
 *
 * @require size() >= 1
 */

public Object head();
PRINCIPLE 5: Add Preconditions Where Appropriate

/**
 * The first element in the queue
 * @require size() \geq 1
 */

public Object head();

It wouldn’t make the software better to pretend that an empty queue has a head element
PRINCIPLE 6: Specify Invariant Properties

/**
 * @invariant size() >= 0
 */

public interface Queue {

PRINCIPLE 6: Specify Invariant Properties

/**
 * @invariant size() >= 0
 */

public interface Queue {

Aim for properties that:
– Help readers build correct conceptual model
– Summarize interaction amongst several methods
Summary 1: The Six Principles

P1. Separate queries from commands

P2. Separate basic queries from derived queries

P3. For each derived query, write a postcondition that specifies what result will be returned, in terms of one or more basic queries

P4. For each command, write a postcondition that specifies the value of every basic query

P5. For every query and command, decide on a suitable precondition

P6. Write invariants to define unchanging properties of objects
Summary 1: The Six Principles

P1. Separate queries from commands

P2. Separate basic queries from derived queries

P3. For each derived query, write a postcondition that specifies what result will be returned, in terms of one or more basic queries

P4. For each command, write a postcondition that specifies the value of every basic query

P5. For every query and command, decide on a suitable precondition

P6. Write invariants to define unchanging properties of objects
A New Basic Query

/**
 * The elements in the queue
 * (first element is at position 0)
 */

public Vector elements();
A New Basic Query

/**
 * The elements in the queue
 * (first element is at position 0)
 */
public Vector elements();

This query does NOT expose the chosen data structure for holding a queue object’s elements.
size() Is Now a Derived Query

/**
 * The number of elements in the queue
 *
 * @ensure return == elements().size()
 */

class Queue {
    public int size();
}


head() Is Also Derived

```java
/**
 * The first element in the queue
 * @require size() >= 1
 * @ensure return == elements().get( 0 )
 */
public Object head();
```
The Command `add()`

```java
/**
 * Add o at the tail of the queue
 * @ensure size() == size()@pre + 1
 * @ensure elements().get( size()-1 ) == o
 */
public void add( Object o );
```
The Command `remove()`

/**
 * Remove the first element from the queue
 *
 * @require size() >= 1
 * @ensure size() == size()@pre - 1
 * @ensure forall int i in 0..size()-1 | elements().get( i ) == elements()@pre.get( i + 1 )
 *
 public void remove();
A Hybrid Command/query

/**
 * Get and remove the first element
 * 
 * @require size() \geq 1
 * @ensure size() == size()@pre - 1
 * @ensure return == head()@pre
 * @ensure forall int i in 0..size()-1 | elements().get( i ) ==
 * elements()@pre.get( i + 1 )
 */

public Object getAndRemove();
public interface QueueFactory {

    /**
     * @ensure return.size() == 0
     */
    public Queue newObjectSatisfyingQueue();
}
Exception in thread "main" java.lang.RuntimeException:  
\src\QueueImplementation.java:65: error:  
postcondition violated (Queue::remove()):  
forall int i in 0..size()-1 |  
elements().get( i ) == elements()@pre.get( i + 1 )  
At QueueImplementation.remove(QueueImplementation.java:407)  
at QueueTester.testAddAndRemove(QueueTester.java:54)  
at QueueTester.run(QueueTester.java:24)  
at QueueTester.main(QueueTester.java:11)
Summary 2

- A class can have a conceptual model presented as a set of basic queries
  - E.g., the elements() of a queue
- Contracts can be expressed in terms of these basic queries
  - Each method can have a precondition and/or a postcondition
  - Each class can have an invariant
- Contracts can be checked at runtime. Checking
  - Ensures code agrees with contract
  - Ensures contract agrees with code
TOPIC 2

Design by Contract and Exceptions
How Best to Start Using Contracts?

- Full contracts involve preconditions, postconditions and invariants
- Working just with preconditions yields immediate benefits (Java™ libraries are full of examples)
- Easy to do: e.g., don’t need to store away "@pre" values necessary in postconditions
- Can manage without a tool such as iContract. Hand-craft contracts using exceptions
Two Cases

• Exception = precondition check
  – Exception thrown if precondition not fulfilled
  – Program with a RuntimeException (no “try/catch” needed)
  – Exception thrown implies bug in calling code

• Exception = test of query
  – Exception signals “query is false”
  – Program with a catchable exception
  – Exception thrown does NOT imply bug in calling code

• Examples should clarify matters!
Case 1: Exception = Precondition Check

Step 1: Define class PreconditionException (subclassof RuntimeException)

Step 2: For each method with a precondition, write this at start of body
        if !(pre) throw
        new PreconditionException(“helpful message”)

Step 3: Write callers that fulfill precondition

Now, if you accidentally write a caller that breaks the precondition, you’ll get the helpful message
Step 1: PreconditionException

import java.lang.RuntimeException;

public class PreconditionException extends RuntimeException {

    PreconditionException(String s) {
        super(s);
    }

}
Step 2: Method With Precondition

/**
 * Remove the first element from the queue
 * Precondition: size() >= 1
 */

public void remove() {
    if !(size() >= 1)
        throw new PreconditionException("Queue:remove:size()>=1");
    //code to remove first element
}
Step 2: Method With Precondition

/**
 * Remove the first element from the queue
 * Precondition: size() >= 1
 */

public void remove() {
    if (!(size() >= 1))
        throw new PreconditionException("Queue:remove:size()>=1");
    // code to remove first element
}

Ideally, tool generates code from comment (or vice versa)
Step 3: Code That Calls remove()

E.g., a method to remove all elements from a queue:

```java
public void removeAllElements(Queue q) {
    while (q.size() >= 1) q.remove();
}
```

If programmer writes an erroneous caller, in which queue has `size()==0` when `remove()` is called, the program will stop and the programmer will see a message that helps with debugging:

```
Exception in thread "main"
    PreconditionException: Queue:remove:size()>=1
```
Alternative: Use assert

/**
 * Remove the first element from the queue
 * Precondition: size() >= 1
 */

public void remove() {
    assert size() >= 1 : "Queue:remove:size()>=1";
    //code to remove first element
}

If size()>=1 is false, the assert throws an AssertionError constructed with string “Queue:remove:size()>=1”
Case 2: Exception = Test of Query

Step 1: Define new subclass of Exception
(can use RuntimeException)

Step 2: For each method with a precondition,
provide a query to test the precondition (optional)

Step 3: At start of method’s body, write
if !(query) throw new xxxException()

Step 4: Write callers that invoke method and test query
in a single call:
– No exception => signals that query was true
– Exception caught => signals that query was false
Step 1: Define QueueEmptySignal

```java
public class QueueEmptySignal extends RuntimeException {
    public QueueEmptySignal() {
        super();
    }
}
```
Step 2a: Method With Precondition

/**
 * Remove first element from queue
 * Precondition: size() >= 1
 */

public void remove() ...
Step 2b: Provide a Query

/**
 * Number of elements in queue
 */

public int size();

-- Don’t always need this, but
-- usually makes life easier
Step 3: Body Throws Exception

```java
public void remove() {
    if(!(size() >= 1)) throw new QueueEmptySignal();
    else
        //code to remove first element
}
```
private void removeAllElements( Queue q ) {
    boolean moreElements = true;
    while (moreElements) {
        try {
            q.remove();
        } catch (QueueEmptySignal s) {
            moreElements = false;
        }
    }
}
Step 4: Caller Tests Precondition

```java
private void removeAllElements(Queue q) {
    boolean moreElements = true;
    while (moreElements) {
        try {
            q.remove();
        } catch (QueueEmptySignal s) {
            moreElements = false;
        }
    }
}
```

The *try/catch* is used to test the precondition on *remove()* that *size()* >= 1
Comparison

```java
while ( q.size() >= 1 ) q.remove();
```
Comparison

“Look before you leap”

while ( q.size() >= 1 ) q.remove();
Comparison

“Look before you leap”

\[
\text{while ( q.size() >= 1 ) q.remove();}
\]
Comparison

“Look before you leap”

while ( q.size() >= 1 ) q.remove();

(Exceptio would signal a bug in the calling code)
Comparison

“Look before you leap”

while ( q.size() >= 1 ) q.remove();

(Exception would signal a bug in the calling code)

while (moreElements) {
    try { q.remove();
    } catch (QueueEmptySignal s) {
        moreElements = false;
    }
}
Comparison

“Look before you leap”
while ( q.size() >= 1 ) q.remove();
(Exception would signal a bug in the calling code)

“Shoot first, ask questions later”
while (moreElements) {
    try { q.remove();
    } catch (QueueEmptySignal s) {
        moreElements = false;
    }

Two method calls to remove each element
Comparison

“Look before you leap”
while ( q.size() >= 1 ) q.remove();
(Exception would signal a bug in the calling code)

“Shoot first, ask questions later”
while (moreElements) {
    try { q.remove();
    } catch (QueueEmptySignal s) {
        moreElements = false;
    }

Two method calls to remove each element

One method call to remove each element
Comparison

“Look before you leap”
while ( q.size() >= 1 ) q.remove();
(Exception would signal a bug in the calling code)

“Shoot first, ask questions later”
while (moreElements) {
  try { q.remove();
  } catch (QueueEmptySignal s) {
    moreElements = false;
  }
(Exception does NOT signal a bug (error, fault, …)
Summary 3: Contracts and Exceptions

• Exceptions are used to implement precondition-checking

• Two modes:
  – Caller is expected to keep to the precondition, else will crash and burn
  – Caller is expected to use an exception to test whether a precondition WAS true

• Both modes have legitimate uses
Final Summary

• Full contracts must be based on a set of basic queries that present a conceptual model of a class

• You can easily achieve weaker (e.g., precondition-only) contracts with handwritten “throw” clauses

• When pre-testing preconditions is too expensive, or open to race conditions, use exceptions as signals

• Runtime checking of contracts helps testing and debugging, and makes documentation trustworthy

• To get full benefit of contracts, you need tool support
If You Only Remember One Thing…

You are now ready to begin using contracts systematically in your programming