Design by Contract for Java - Revised

Master’s thesis - Johannes Rieken

May, 9th 2007
Outline

• Introduction: Design by Contract (DBC)
• Existing approaches:
  ‣ “How to specify / validate / enforce contracts?”
  ‣ Summary of existing approaches
• A new approach to implement DBC for Java
  ‣ Fundamentals: Java 5 Annotations, Annotation Processing API, Java Compiler API, Instrumentation API
  ‣ “How to specify / validate / enforce contracts? - Revised”
  ‣ Demo
  ‣ Summary
Design by Contract (DBC)

- Brainchild of Bertrand Meyer, firstly implemented in Eiffel
- Basic idea: Improve software reliability by defining contracts between interacting software components
- When invoking a method, contracts state which **pre-conditions** a client must fulfil, and what **post-conditions** a supplier guarantees
- Further concepts: Invariants, Side-effect freeness, Pre & Post-state, Inheritance of contracts

```java
public double sqrt(int n){
    pre n >= 0;
    ...
    post (result * result) == n;
}
```
DBC in Java

• Java does not support DBC - except for the “simple assertion facility”
• Allows to define assertions with the `assert` keyword
• Full support for DBC remains the most requested enhancement
• Numerous tools and extensions exist:
  - Java Modelling Language (JML), Iowa State University
  - Jass (Java with Assertions), Universität Oldenburg
  - jContractor, Contract4J, C4J, OVal, AOP
  - ...
• Still, DBC can only rarely be seen in the Java World
DBC-Approaches: Requirements

- **Functional Requirements:**
  - How to specify contracts?
  - How to validate contracts?
  - How to enforce contracts?

- **Non-functional Requirements:**
  - Ease-of-use (Specify and validate contracts easily)
  - Seamless integration into IDEs and build processes
How to specify contracts? (1/2)

- Any kind of Java comment

```java
/**
 * @pre o != null
 */
public void add(Object o) { ... }
```

- Java Method, AOP-Aspect

```java
@Contract( bind = BufferContract.class)
public Object last() { ... }
```

- Java 5 Annotations

```java
@NotNull
public void add(Object o) { ... }

@Post(expr = "@Result != null")
public Object last() { ... }
```
# How to specify contracts? (2/2)

<table>
<thead>
<tr>
<th></th>
<th>Java Comments</th>
<th>AOP-Aspects / Java Methods</th>
<th>Java 5 Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Possible targets</strong></td>
<td><strong>everywhere</strong></td>
<td><strong>targets methods</strong></td>
<td><strong>types, fields, methods, parameters, packages</strong></td>
</tr>
<tr>
<td><strong>Bytecode representation</strong></td>
<td><strong>no, pre-processor or compiler is obligated</strong></td>
<td><strong>yes, but without target</strong></td>
<td><strong>yes</strong></td>
</tr>
<tr>
<td><strong>Processing tools</strong></td>
<td><strong>Doctlet API (limits targets)</strong></td>
<td><strong>AOP-Tools, Java compiler</strong></td>
<td><strong>Annotation Processing Tools</strong></td>
</tr>
<tr>
<td><strong>Notes</strong></td>
<td>Requires parsing as Java comments are Strings only, no tool support!</td>
<td>a “contract method” or aspect must be assigned to the right method</td>
<td>Possibly used as a “String-container” which requires further parsing</td>
</tr>
</tbody>
</table>
Validate & enforce contracts

- Once a contract has been specified, it
  - should be validated like any other Java expression (optional, but desired)
  - must be enforced at runtime
- Different technical approaches exists:
  - Pre-Processor
  - Bytecode Instrumentation
  - Custom Compiler
  - Aspect Oriented Programming
  - Interpreter
Perform sourcecode to sourcecode transformation

Involves (a) a parser for the actual programming language and (b) a parser for the assertions, (c) source-code rewriter

✓ Easy to understand and straightforward to implement

- High maintenance costs as Java evolves, writing a good parser it not trivial
- Tool chain which is hard to integrate, e.g. Eclipse, NetBeans
- Errors puzzle users, because the executed code differs from the written code

// * @ pre o != null */
void add(Object o){ ...

Pre-Processor

void add(Object o){
  if(! (o != null) ){ throw new Error(); }
  ...
}
Validate & Enforce contracts:
Bytecode Instrumentation

- "Weave" contract method into target method (bytecode 2 bytecode transformation)

✓ Bytecode transformation can be performed transparently

- Where does the bytecode for contracts come from?
- Dynamic bytecode instrumentation is slow - esp. when bytecode is generated on-the-fly
- Static bytecode instrumentation requires an addition processing step
Validate & Enforce contracts:

Custom Compiler

- Sourcecode to bytecode transformation
- Involves (a) a parser for the actual programming language and (b) a parser for the assertion, and a bytecode generator

✓ Most natural way to integrate DBC into Java
✓ No tool chain or alike
- Extremely high engineering and maintenance costs
- Users tend to stick to the default compiler

```java
/**
 * @pre o != null */

void add(Object o){ ... }
```
1. Aspect Oriented Programming (AOP) to weave contracts into code
   • contracts is either specified as aspect, or
   • aspect is generated by a pre-processor from a contract
   ✓ DBC is a cross-cutting-concern
   - AOP is not Java!

2. Interpreter Approach evaluates contracts on-the-fly
   ✓ Requires no pre-processing (e.g. compiler, bytecode instrumentation), but uses a evaluation engine like JRuby or Groovy
   - Evaluation errors because of invalid contracts
   - Some evaluation engines differ slightly from Java syntax
Summary: Existing Approaches

- Approaches, introduced so far, have drawbacks
  - High engineering or maintenance costs (pre-processor, compiler)
  - Hard to integrate into IDEs or build processes (pre-processor)
  - Different programming paradigms (AOP)
  - Lack of ease of use (Comment-based tools, Interpreter)

✓ Solution: Utilise enhancements that came with Java 5 and Java 6
  - Java 5 Annotations, Annotation Processing API (JSR 269)
  - Compiler API (JSR 199)
  - Instrumentation API
Basics: Java 5 Annotations

• An annotation is a “type-safe comment” build for automatic processing

• Annotations are types, similar to Java interfaces but more restricted

@interface Todo {
  String subject();
  float duration() default 1;
}

Definition

@Todo(
  subject = "Implement method!",
  duration = 2.6
) public void foo(Object bar){ }

Application

• may define attributes which store compile-time constants

• data type of attributes must be a primitive, a string, java.lang.Class, an enum-type, other annotations, or an array of these types

• the meta-annotation java.lang.annotation.Target allows to restrict possible targets (e.g. methods only)
Basics: Annotation Processing API

- Evolved from JSR 269 and added in Java 6 (forerunner is the annotation processing tool from Java 5)
- Allows to write plug-ins for the Java compiler
- Allows to issue error and warning messages
- Provides a model to access the program currently being compiled
- Easy to integrate: Every annotation processor, that can be found on the classpath, participates in the compilation process by default
- Tools supporting JSR 269: Javac, NetBeans, Idea, and Eclipse (3.3)
Basics: Compiler & Instrumentation API

- **Compiler API** evolved from JSR 199 and was added in Java 6
- Allows to interact with the Java compiler on a high level
- Source & Bytecode files are represented as file objects which enables arbitrary implementations, esp. in-memory source files
- Reporting-Infrastructure for error and warning messages

- **Instrumentation API** leverages support for bytecode instrumentation
- A Java-agent is loaded by the JVM prior to a program, and
- can instrument bytecode while the program is loaded into the JVM
- Easy to integrate: Use the `-javaagent` switch of the `java` program
Novel Approach

• Functional Requirements:
  • How to specify contracts? - *Java 5 Annotations*
  • How to validate contracts? - *Annotation Processing & Compiler API*
  • How to enforce contracts? - *Bytecode Instrumentation & Compiler API*

• Non-functional Requirements:
  • Ease of use (Specify and validate contracts easily) - *Standard Java*
  • Seamless integration into IDEs and build processes - *Standard Java*
Specification Annotations

- @Invariant
- @Model, @Represents
- @Pre, Post
- @SpecCase, @Also
- @Pure
- @NonNull, @Min, @Max, @Range, @Length
- Specification Expressions, not annotations: @Result, @Signal, @Old, @ForAll, @Exists

Abstract class Foo {
    @Invariant("bar != null") Object bar;
}

@Also{
    @SpecCase( pre = "o != null", post = "contains(o)" ),
    @SpecCase( pre = "o == null", signals = NPE.class )
}
void add(Object o){ ... }

void openValve(@Range(from = 0, to = 100) int n) { ... }

@NonNull Object get(){ ... };
Validate Contracts *(compile-time)*

1. Be an annotation processor & have a model reflecting the structure of the program

2. Create contract methods - the body of a contract method is the content of an annotation

3. Compile the transformed model

4. Link errors and warnings in contract methods with the corresponding annotation

5. Annotation Processing API to report messages
Validate Contracts (compile-time)

1. Be an annotation processor & have a model reflecting the structure of the program
2. Create contract methods - the body of a contract method is the content of an annotation
3. Compile the transformed model
4. Link errors and warnings in contract methods with the corresponding annotation
5. Annotation Processing API to report messages
Enforce Contracts *(runtime)*

1. Be a Java agent!
2. Create & transform model in the same manner as during contract validation
3. Compile transformed model and keep the generated bytecode
4. Weave the contract bytecode into the running program

๏ Why are we re-doing steps 1-3?
Enforce Contracts *(runtime)*

1. **Be a Java agent!**

2. **Create & transform model** in the same manner as during contract validation

3. **Compile transformed model and keep the generated bytecode**

4. **Weave the contract bytecode into the running program**

- **Why are we re-doing steps 1-3?**
How it looks
Demo
Features: Modern Jass

- Method Specifications & Invariants (@SpecCase, @Also, @Invariant)
- Model variables (@Model, @Represents)
- Full support for inheritance of contracts
- Flyweight specifications (@NonNull, @Min, ...)
- Supports the notion of side-effect freeness (@Pure)

- Integrates seamlessly: Annotation Processor, Java Agent
- Low maintenance costs: Compiler API
Limitations (Concept)

• Annotations & Annotation Processing:
  ‣ limited set of annotation targets, e.g. no annotation for loops
  ‣ waiting for JSR 305, JSR 308
  ‣ Processing for annotated types only

• Bytecode:
  ‣ Missing debug information (parameter names)
  ‣ Line-Number-Table should contain more information

• Implementation of the concept has limitations, too
Future Work

• Further collaboration with the JML 5 project - unified set of specification annotations

• Improve tool implementation (remove bugs, improve speed)

• Open source everything ( http://modernjass.sourceforge.net )

• Investigate different implementation approaches, esp. reusing some Eclipse or NetBeans components

• Watch out for upcoming changes: JSR 305, JSR 308, Eclipse: Code-completion for annotation processor, syntax-highlighting in attribute-values
Summary

• The chosen concept is applicable for DBC!
• Rich feature set (can be compared to JML)
• “Ease of use” and “Ease of integration” (that is something new)
• Prototypic implementation could be used to realise different case studies
• Still, prototype is not for productive use (in parts only)
Thanking you for your kind attention!