Karlsruhe Institute of Technology

Programmierparadigmen

Prof. Dr. Ralf Reussner

Topic 6

Design by Contract

DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY, FACULTY OF INFORMATICS

dsis.kastel.kit.edu

Overview on Today's Lecture

- Content
  - Motivation for design by contract
  - Formal foundation: Hoare Triples
  - Contracts for methods
  - Programming contracts with JML
  - Contract checking and tools
- Learning goals: participants –
  - understand what design by contract is
  - can argue the benefits of design by contract
  - are able to understand a contract and identify if and where it is violated
  - know about languages and tools for defining, validating and verifying contracts
  - are able to define simple contracts in JML

A Simple Stack Class

- Is this class correct?

```
class Stack {
    private Object[] elements = new Object[10];
    private int size = 0;
    void push(Object element) {
        this.elements[size++] = element;
    }
    public Object pop() {
        return this.elements[--size];
    }
    public Object top() {
        return this.elements[size-1];
    }
}
```

- Is this usage correct?

  You may argue that it is not, *but why?*

```
Stack myStack = new Stack();
myStack.push(new Object());
myStack.pop();
myStack.pop();
```

Software Correctness

- What is software?
  - Artefacts created during the software development process
  - E.g. specifications, design-documents, programs
- What is *correct* software?
  - Software fulfilling its specification
    → **Correctness is a relative notion**
  - “Software verified against its specification”
  - In our previous example, there was no specification
    (although you probably had one in mind)
- Validation and verification
  - Validation: Test the software with exemplary inputs (covering the most relevant equivalence classes of input data)
  - Verification: Prove software correctness for all possible input data (universal quantification)

What is the Specification of `Stack`?

- Informally: Manipulation of a data structure with pop and push operations according to the LIFO principle
- Additional information is necessary:
  - What if the stack is empty when calling `pop` or `top`?
  - What if the stack is full when calling `push`?
  - What if `null` is tried to be pushed?
  - What happens if several operations are called simultaneously?
  - etc.

→ Informal specifications via documentation and method/parameter naming are often incomplete, inconsistent, contradictious and can only be manually checked

- A formal specification allows automated correctness checks, but is difficult to define and expensive

Specifications via Contracts

- Contracts are defined between a supplier and a client of a service about their responsibilities
- Participants:
  - The supplier of a service
    - In our example the class `Stack`
  - The client using a service
    - In our example the code calling operations of the class `Stack`
- Responsibilities:
  - Who has to do what?
  - Under which circumstances?

Formal Foundation: Hoare Triples

- Hoare triples are a central feature of the Hoare logic
  - A formal system for checking semantic correctness
  - Proposed by Sir Tony Hoare in 1969
- Form of a Hoare triple: {P} C {Q}
  - P: precondition
  - C: series of statements
  - Q: postcondition
- Semantics: If P is true before the execution of C, then Q is true after executing C
- Trivial example: {x >= 9} x := x + 5 {x >= 13}

Contracts for Methods

- In software, contracts can be used to specify the semantics of methods
  - Precondition: Specification what the supplier expects from the client
  - Statements: The method body
  - Postcondition: Specification of what the client can expect from the supplier *if the precondition is fulfilled*
- Contracts do not only specify the behavior of a method but also how they have to be used by a client
  - Client has to ensure that the precondition is fulfilled
  - Client has to correctly handle the guaranteed result
- Design by contract idea from Bertrand Meyer [Meyer1992, Meyer1997]
  - Contractual use of methods
  - The supplier guarantees the postcondition if the precondition is fulfilled (by the client)
  - Implemented in the *Eiffel* programming language

Preconditions and Postconditions for `Stack`

- Preconditions:
  - `push` may not be called if the stack is full
  - `pop` / `top` may not be called if the stack is empty
- Postconditions:
  - After calling `push`, the stack may not be empty, the top element is the one that was pushed and its number of elements was increased by one
  - After calling `pop`, the stack may not be full and its number of elements has been decreased by one
  - After calling `top`, nothing has to be changed
- Are these pre- and postconditions complete?

```java
public void push(Object element) {
    size++;
    for (int i = 0; i < size; i++) {
        this.elements[i] = element;
    }
}
```

This code would also fulfill the specified postcondition

Obligations and Benefits

- Method `push` example:

	Obligations	Benefits
Client	*Satisfy precondition:* Only call `push(x)` on a non-full stack	*From postcondition:* Get stack updated: not empty, x on top, size increased by one
Supplier	*Satisfy postcondition:* Update stack representation to have `x` on top, `size` increased by one, not empty	*From precondition:* Simpler processing thanks to the assumption that stack is not full

adapted from [Meyer1997]

- With the precondition that the stack must not be full when calling `push`, the `Stack` class must not deal with that case
  - Alternatively, the precondition could be relaxed, but then the behavior of the stack in that case would have to be specified

Precondition Design and Non-Redundancy

Non-redundancy principle [Meyer1997]
The body of a routine shall not test for the routine's precondition.

- Often, code contains assertions or checks the precondition and throws exceptions if they are not fulfilled
- Defining contracts and checking the code against them makes those additional checks obsolete
- **Precondition design**:
  There are two possibilities to deal with a precondition for a method:
  1. Demanding: Assign the responsibility to the client by adding the condition to the routine's precondition. Than the client needs to deal with cases where the pre-condition is not met.
  2. Tolerant: Assign the responsibility to the supplier by adding a conditional instruction and handling of violations to the control flow of the method body
- The non-redundancy principle demands that the condition enforcement should only be assigned to either the client or the supplier

Abstraction and Intent

- Contracts abstract from concrete implementations
  - The implementation is irrelevant for a service signature and its contract
  - E.g. a square root calculation can be implemented using linear or binary search, newton's method etc.
- Different implementations can have varying non-functional properties (efficiency, memory usage etc.) but provide equal functionality

- Code itself also defines an implicit contract, but a poor one
  - No separation between contract and implementation decisions
  - Once the implementation changes, the contract might change unintendedly, too.
- Explicit contracts have several benefits
  - Allow to specify intent
  - Allow to change implementation details
  - Require a client to only understand the contract, not the implementation
  - Can be used to verify the code against them

Programming Contracts

- Several programming languages support design by contract –
  - either with pre- and postconditions as first-level entities
  - or as language extensions
- Different degrees of design by contract support:
  - Documentation: Contracts are purely informative
  - Validation: Contracts are used to generate test cases to validate if the code works as expected
  - Verification: Contracts are used to statically check if the code fulfills them
- Examples for tools and languages:
  - Eiffel: programming language with integrated contracts (by Bertrand Meyer)
  - Object Constraint Language (OCL): specialized constraint language, especially used to define contracts in the UML
  - Java Modeling Language (JML): language for specifying contracts in Java

Programming Contracts: Eiffel

- Design by contract is fully implemented in the Eiffel language
- Method pop example:

```
pop(): Object is
    require
        size > 0
    do
        … pop logic …
    ensure
        size = old size - 1
    end
```

References the value of an expression before the method was executed

- `require` and `ensure` define the pre- and postcondition
- The example contracts lacks:
  - A guarantee about the non-modification of the rest of the stack
  - A guarantee about the identity of the returned element

Java Modeling Language (JML)

- Formal behavioral interface specification language for Java
- Based on the design by contract idea
- Pre- and postconditions can be defined in specialized Java comments (all lines starting with `@`) in extended Java expression syntax
- Essential keywords are `required` for preconditions and `ensures` for postconditions
- Simple example for the pop method:

```
/*@ requires size > 0;
  @ ensures  size == \old(size) – 1;
  @*/
Object pop() { … pop logic … }
```

Basic JML Syntax (1)

- Syntax extensions to Java expressions:

Syntax	Meaning
a ==> b	a implies b
a <==> b	a iff b
a <=!=> b	!(a <==> b)
\result	Result of the method call
\old(E)	Value of E in the state before method execution

Only available in postconditions

- Extended contract for pop method:

```
/*@ requires size > 0;
  @ ensures  size == \old(size) - 1 && \result == \old(top());
  @*/
Object pop() { … pop logic … }
```

It is possible to combine conditions with either the *and* operator or by starting a new `requires` / `ensures` lines

Basic JML Syntax (2)

- It would also be possible to specify that pop returns `null` if the stack is empty:

Weakest kind of precondition:
Can be omitted

```
/*@ requires true;
  @ ensures  size  > 0 ==> \result == \old(top());
  @ ensures  size <= 0 ==> \result == null;
  @*/
Object pop() { … pop logic … }
```

Quantifiers in JML

- JML provides universal and existential quantifiers:
  - **(\forall** *declaration*; *range-expression*; *body-expression*)
  - (**\exists** *declaration*; *range-expression*; *body-expression*)
- We can now ensure that pop does not change other stack elements:

```
/*@ requires size > 0;
  @ ensures  size == \old(size) - 1
  @ ensures  \result == \old(top())
  @ ensures  (\forall int i; 0 <= i && i < size;
                 \old(elements[i]) == elements[i]);
  @*/
Object pop() { … pop logic … }
```

- There are also quantifiers `\max`, `\min`, `\sum` etc.

JML: Side Effect Restrictions

- JML provides further keywords to restrict allowed side effects:
  - `assignable`: Specifies fields that are allowed to be written by the method
    - Per default, every field is assignable
    - The opposite of the default case can be specified as: `assignable \nothing`
  - `pure`: a method declared as pure has no side effects
    - i.e. the method does not modify any data
    - Implies `assignable \nothing`
- We can use that to define that calling top has no side effects:

```
/*@ requires size > 0;
  @ assignable \nothing;
  @*/
Object top() { … top logic … }
```

We could also use @pure

  - Makes postconditions on fields (especially the `elements` array and `size`) obsolete

Exceptions

- JML allows to specify expected exceptions
  - `signals`: defines a postcondition for the case that an exception is thrown
    - Signature: `signals (E e) P`
    - If an exception of type `E` is thrown, the predicate `P` must be true
    - Only exceptions defined via signals are allowed to be thrown
    - Only defines that an exception *may be* thrown, not that is *must be* thrown
  - signals_only: defines which exceptions may be thrown by the method
    - Signature: `signals E1, E2, …`
    - A short form for `signals` without defining a condition
    - `Signals` implies `signals_only` without additional conditions
- Expecting a `NullPointerException` if pushing `null` on the stack:

```
/*@ requires size < 10;
  @ signals  (NullPointerException npEx) element == null;
  @ ensures …
  @*/
void push(Object element) { … push logic … }
```

Does **not** mean that an exception is thrown if the input value is `null`!

Class Invariants

- Class invariants define conditions of the containing class that must hold in all user visible states
  - Must hold after each constructor execution (constructor postcondition)
  - Must hold before and after each method execution (method pre- and postcondition)
- Allow to define acceptable and consistent object states

```
class Stack {
   //@ invariant size >= 0 && size <= 10;

   private Object[] elements = new Object[10];
   private int size = 0;
   …
}
```

Class Correctness

- We can finally define when a class `C` is correct:
  - Let $\mathtt{INV}_\mathtt{C}$ be the invariants of class `C`
  - Let $\mathtt{CONSTR}_\mathtt{C}$ be the constructors of class `C`
  - Let $\mathtt{METHODS}_\mathtt{C}$ be the methods of class `C`
  - Let $\mathtt{PRE}_\mathtt{M}$ be the preconditions, $\mathtt{BODY}_\mathtt{M}$ the operation body and $\mathtt{POST}_\mathtt{M}$ the postconditions of a method `M`

Class `C` is correct

$$\Leftrightarrow$$

$$\forall\ \mathtt{I} \in \mathtt{CONSTR}_\mathtt{C} : \{\ \mathtt{PRE}_\mathtt{I}\ \}\ \mathtt{BODY}_\mathtt{I}\ \{\ \mathtt{POST}_\mathtt{I}\ \mathtt{and}\ \mathtt{INV}_\mathtt{C}\ \} \quad \textit{and}$$
$$\forall\ \mathtt{M} \in \mathtt{METHODS}_\mathtt{C} : \{\ \mathtt{PRE}_\mathtt{M}\ \mathtt{and}\ \mathtt{INV}_\mathtt{C}\ \}\ \mathtt{BODY}_\mathtt{M}\ \{\ \mathtt{POST}_\mathtt{M}\ \mathtt{and}\ \mathtt{INV}_\mathtt{C}\ \}$$

  - To be precise, it would also necessary to demand that all fields of class `C` have default values in the precondition of the constructors

Liskov Substitution Principle

- The Liskov substitution principle restricts the possible pre- and postconditions of an overwriting method
  - Preconditions must not be more restrictive than those of the overwritten method: $\texttt{Precondition}_{\texttt{Super}} \Rightarrow \texttt{Precondition}_{\texttt{Sub}}$
  - Postconditions must be at least as restrictive as those of the overwritten methods: $\texttt{Postcondition}_{\texttt{Sub}} \Rightarrow \texttt{Postcondition}_{\texttt{Super}}$
- Regarding a complete class, the following rules apply:
  - Pre- and postcondition relations must hold for all methods as stated above
  - The class invariants must be at least as restrictive as those of the superclass: $\texttt{Invariants}_{\texttt{Sub}} \Rightarrow \texttt{Invariants}_{\texttt{Super}}$
- A special case of this is known from parameter and return types of methods (Co-/Contravariance)

Liskov Substitution Principle: Example

```
class Stack {

    //@ requires element != null;
    //@ ensures …
    void push(Object element) {
        // … push logic …
    }

    //@ requires …;
    //@ ensures \result != null;
    Object pop() {
        // … pop logic …
    }
}
```

```
class NullAcceptingStack
    extends Stack {

    //@ requires true;
    //@ ensures …
    void push(Object element) {
        // … push logic …
    }

    //@ requires …;
    //@ ensures true;
    Object pop() {
        // … pop logic …
    }
}
```

- Does this code fulfill the Liskov substitution principle?

  No, because the postcondition of pop gets weaker in the subclass

Precondition Availability

Precondition availability rule [Meyer1997]
Every feature appearing in the precondition of a routine must be available to every client to which the routine is available.

- With this rule, every client is able to check the precondition of a routine
- This rule does not apply to postconditions
  - Postconditions can reference features that are not available to a client
  - Such postconditions are not usable by clients
  - But be careful: There is a risk to define postconditions on internal behavior, which reduces the changeability of implementation details

Precondition Availability: Example (1)

- Recall our stack implementation and our contracts for it:

```
class Stack {
   private Object[] elements = new Object[10];
   private int size = 0;

   /*@ requires size > 0;
     @ ensures  size == \old(size) - 1
     @ ensures  \result == \old(top())
     @ ensures  (\forall int i; 0 <= i && i < size;
                         \old(elements[i]) == elements[i]);
     @*/
   Object pop() { … pop logic … }

   …
}
```

Be aware that this is also internal state

- Does this implementation adhere to the precondition availability rule?

  No, because `size` is not accessible by the client, so we cannot determine whether the stack is full or empty

Precondition Availability: Example (2)

- Simple solution: add a method for retrieving the number of elements

```
class Stack {
    private Object[] elements = new Object[10];
    private int size = 0;

    /*@ requires getElementCount() > 0;
      @ ensures  getElementCount() == \old(getElementCount()) – 1
      @ ensures  \result == \old(top())
      @ ensures  (\forall int i; 0 <= i && i < getElementCount();
                        \old(elements[i]) == elements[i]);
      @*/
    Object pop() { … pop logic … }

    public int getElementCount() {
        return size;
    }

    …
}
```

- Another possibility is to define `isEmpty` and `isFull` methods

Non-pure Methods in Contracts

- It is not allowed to reference methods causing side effects in contracts
  - Evaluating the condition would result in a state change of the object

    → Our newly defined method (and also the `top` method) must be free of side effects
  - Referenced methods must be defined as `pure`:

```
public /*@ pure @*/ int getElementCount() {
    return size;
}
```

- Declare methods as pure / functional as often as possible

Null-Values in JML

- JML can only interpret expressions when their assertion does not cause exceptions and yields the value „true“
  - ➔ Dereferencing `null`, problematic
  - ➔ Default Case in JML: reference types cannot be `null`
- JML introduces the `nullable` and `non_null` annotation for declarations whose type is a reference type
- Declarations are implicitly `non_null` by default unless they are annotated with the `nullable` modifier
- Possibility of `nullable` by default by annotating the outer most class or interface with `nullable_by_default`

```
/*@ requires true;
  @ ensures  size  > 0 ==> \result == \old(top());
  @ ensures  size <= 0 ==> \result == null;
  @*/
/*@ nullable @*/ Object pop() { … pop logic … }
```

Internal Data in Contracts

- JML provides two further ways to reference private fields in contracts:
  1. Specify the field as public only for the specification:

     ```
     private /*@ spec_public @*/ int size = 0;
     ```

  2. Specify the condition as `private behavior`

     ```
     /*@ ensures \result != null;
       @ private behavior
       @ requires size > 0;
       @*/
     Object pop() { … pop logic … }
     ```

     Every expression that follows this line belongs to `private behavior`

     - This is also necessary for postconditions referencing private fields
     - Also `protected behavior` can be defined
- **Attention:** Examples prior to this slide may not be used in an analysis due to missing elements such as `private behavior or nullable-`Annotations

Checking Contracts

- Contracts can be used to define a specification
- Contracts can additionally be used to check code against them if they are specified in a formal language
- Different ways of checking:
  - Perform a static contract check / verification (no code execution)
  - Perform a dynamic contract check (convert contracts into runtime checks / assertions)
  - Generate test cases from the specification

Checking JML Contracts

- For JML, several tools for checking contracts exist
  - Static contract checking, e.g. using KeY [Beckert2007] or `jml` [JML]
  - Runtime contract checking, e.g. via generated Java assertions in `jmlc` [Bhorkar2000]
  - Testing, e.g. via generated Junit test in `jmlunit` [Cheon2002]
- OpenJML (http://www.openjml.org/)
  - Support for static (`esc`) and runtime checking (`rac`)
  - Integrates popular logic solvers for static checking
  - Is available as command-line tool but also has an Eclipse integration and can be installed from an updatesite
  - Unfortunately, it is not capable of statically verifying our Stack example
- For details on syntax checking with OpenJML, please take a look at the appendix

Assertions as Contracts

- The Java `assert` keyword can be used for simulating contracts
  - Syntax:
    ```
    assert expression1;
    assert expression1 : expression2;
    ```
  - If `expression1` is not true, an exception containing the value of `expression2` is thrown
  - Method `push` example:

```
Object pop() {
    assert size > 0;
    int _oldSize = size;
    Object _oldTop = top();
    // … pop logic …
    assert size == _oldSize - 1;
    assert result == _oldTop;
    return result;
}
```

References to old values have to be stored before executing the method logic

Assertions as Contracts: Discussion

- Assertions can be used to check contracts at runtime
  - However, they are bad for specification purposes
  - They are restricted to Java expressions
- Assertions are not control structures
  - Conditional instructions (if-then-else) are appropriate control structures
  - Assertions only check a contract that must always hold during runtime

Assertion violation rule [Meyer1997]
A runtime assertion violation is the manifestation of a bug in the software.

  - A violated precondition is the manifestation of a bug in the client
  - A violated postconditions is the manifestation of a bug in the supplier, if the precondition is precise (In case of weak preconditions the bug may also be in the client and slipped through the weak precondition)
- Assertions are not an input checking mechanism
  - Assertions have to hold for software-to-software communication
  - They cannot be used to check user inputs

Contracts in Software Modeling

- Contracts are useful to define the specification of a software in early development phases, e.g. during design
- Object Constraint Language (OCL) by OMG
  - Supports the definition of contracts (specialized for models)
  - Can, for example, be used in the UML for behavior descriptions, such as method contracts in class diagrams
  - Requires the definition of a `context` (the method) and the pre- and postconditions after the keywords `pre:` and `post:`

```
context Stack::pop(): Object
   pre: self.size > 0
   post: result = top@pre() and size = size@pre
   body: // … pop logic …
```

References the expression value before method execution

Design by Contract: Discussion

- Performance impact
  - Checking contracts during execution is costly (non-performant)
  - Should not fail in a released product → only execute them in debug mode
  - Contracts are not for error handling, but only report bugs
- Degree of detail
  - If a contract is complete for a (sub)program, it provides a functional specification of that (sub)program (analogue to a program written with a functional programming language)
  - No further program logic would be required, but the program could be generated from the specification
    → shift to a more abstract (functional) programming paradigm
- Contracts are most useful when a postcondition can be given that is simpler than the code computing the result (as otherwise bugs are as likely in the postcondition as in the code)
  - Example:
    (better epsilon comparison)

```
//@ ensures \result * \result == a
double sqrt(double a)
```

Design by Contract: Benefits & Limitations

- Benefits
  - Developing less error-prone software through –
    - better documentation
    - reduced error-handling code
    - clear responsibilities
    - results of static and runtime checking as a debugging aid
  - Possibility to verify an implementation against a formal specification (which can be necessary for certifying software)

- Limitation: Protocols
  - Complex conditions that are as error prone as the method body itself
  - Design by contract does not allow directly to define protocols, such as "routine A has to be executed before routine B"
  - Protocols can be realized using status variables, requiring increased development and maintenance effort

Conclusion

- Design by contract is a principle for the contractual use of methods
  - Explicit pre- and postconditions for methods
  - Based on Hoare triples (precondition, statements, postcondition)
  - Originally implemented in the Eiffel language
  - Formally defined contracts can be used for verification purposes
- Java Modeling Language (JML)
  - Specialized Java comments with a functional software specification
  - Allow to formalize behavior of Java classes and methods
  - Tools such as OpenJML provide static or runtime checking of contracts
- Benefit: Less software bugs through –
  - Clear responsibilities
  - Better documentation
  - Automated checking and verification

Literature and References

[Meyer1992]	Bertrand Meyer. Applying "Design by Contract". Computer, 25, 40-51, 1992
[Meyer1997]	Bertrand Meyer. Object-oriented Software Construction. Prentice Hall, 1997
[Beckert2007]	Bernhard Beckert, Reiner Hähnle, Peter H. Schmitt. Verification of Object-Oriented Software. The Key Approach. Springer, 2007
[Bhorkar2000]	Abhay Bhorkar. A Run-time Assertion Checker for Java using JML. Department of Computer Science, Iowa State University, TR #00-08, 2000
[Cheon2002]	Yoonsik Cheon, Gary T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In: ECOOP 2002 — Object-Oriented Programming. Lecture Notes in Computer Science, Vol. 2374. Springer, Berlin, Heidelberg, 2002.
[JML]	http://www.eecs.ucf.edu/~leavens/JML/index.shtml

APPENDIX

Manual (Static) Contract Checking

- Where is this contract violated?

```
public class AddTest {
    //@ requires b > 0;
    //@ ensures a >= 0 ==> \result == a-b;
    //@ ensures a < 0 ==> \result == 0;
    public int add(int a, int b){
        if (a < 0) return 1;
        return a-b;
    }
    public static void main(String args[]) {
        System.out.println(new AddTest().add(2,-1));
    }
}
```

Method does not fulfill postcondition

Caller does not fulfill precondition b > 0

- General contract checking:
  - Check if all callers fulfill the preconditions
  - Check if the method fulfills its postconditions (provided that the preconditions are fulfilled)

Static Contract Checking with OpenJML

- OpenJML yields two warnings in this example:

```
public class AddTest {
    //@ requires b > 0;
    //@ ensures a >= 0 ==> \result == a-b;
    //@ ensures a < 0 ==> \result == 0;
    public int add(int a, int b){
        if (a < 0) return 1;
        return a-b;
    }

    public static void main(String args[]) {
        System.out.println(new AddTest().add(2,-1));
    }
}
```

Cannot establish an assertion for this postcondition

Cannot establish an assertion: precondition is false

- To execute OpenJML with the Z3 solver [1] on the example program:

```
java -jar $OJ/openjml.jar -exec $Z3/z3.exe -esc AddTest.java
```

Path to the OpenJML folder

Path to the Z3 solver folder

Runs the static checker

[1] http://www.openjml.org/downloads/

Dynamic Contract Checking with OpenJML

- Executes the program with runtime checks
  - Per default, violations are reported but the program proceeds
  - Several options provide other behavior
- The dynamic contract checking requires two steps:
  1. Compile the program with the `rac` option of OpenJML:

     ```
     java -jar $OJ/openjml.jar -rac AddTest.java
     ```

     Path to the OpenJML folder

     Adds additional statements for runtime checking

  2. Execute the program with OpenJML added to the classpath:

     ```
     java -cp $OJ\openjml.jar:. AddTest
     ```

     Path to the OpenJML folder

     Replace ":" with ";" on Windows systems

- Runtime checking will only yield violations that occur in the executed code, so there is no verification of conditions