Design by Contract

Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

ADAP C04

Agenda

1. Design by contract
2. Expressing contracts
3. Realizing contracts
4. Contract violations
5. Contracts in context

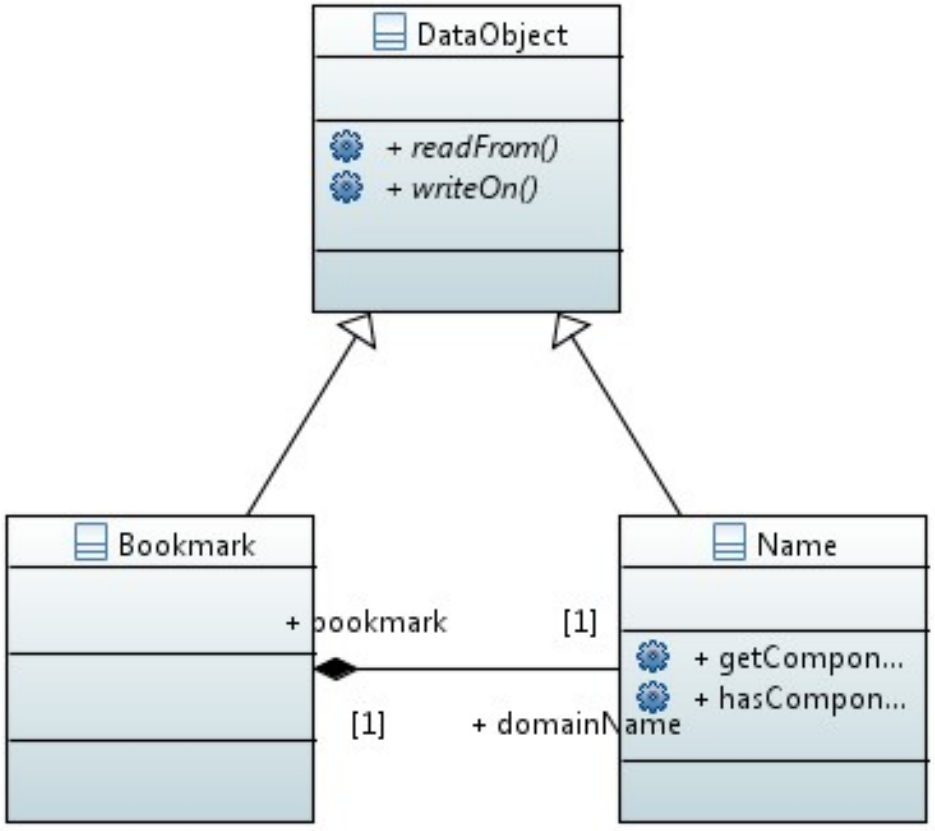
1. Design by Contract

Design by Contract

Design by contract views software design as a succession of contracting decisions [M91].

Design by Contract is method for specifying interfaces as contracts to clarify responsibilities and minimize programming effort while improving reliability [DR].

Classes Specify Object Collaborations



Contracts

- A contract specifies rights (benefits) and obligations
  - Between a client (consumer) and contractor (supplier)
  - Contracts are (ideally) exhaustive; there are no hidden clauses
- Rights and obligations are mutual
  - A client obligation (precondition) is contractor right
  - A contractor obligation (postcondition) is a client right
- The contract protects both sides of the deal
  - The client is guaranteed a result
  - The contractor is guaranteed a specified operating environment
- **Contracts are turned into interface specifications**

The AbstractName#insert(...) Method

```
public void insert(int i, String c) {
  assertClassInvariants();

  assertIsValidIndex(i, getNoComponents() + 1);
  assertIsNonNullArgument(c);
  int oldNoComponents = getNoComponents();

  doInsert(i, c);

  assert (oldNoComponents + 1) == getNoComponents() : "..."; // [1]

  assertClassInvariants();
}

protected abstract void doInsert(int i, String c);
```

[1] Please note that assertion checking using the assert keyword can be switched off; if you require the assertion be checked, do not use assert, but a classic if clause.

Contract for Name#insert(...)

	Rights	Obligations
Client	Receives Name object with component inserted	Ensures defined environment, i.e. index is valid and component != null
Contractor	Operates in defined environment	Provides Name object with component inserted

Defensive Programming

- Defensive programming
  - Wikipedia: “[...] the programmer never assumes a particular function call or library will work as advertised”
  - Meyer: “[...] protect every software module by as many checks as possible, even those which are redundant with checks made by the clients.”
- Problems with defensive programming
  - Multiplies the amount of checking code
  - Leads to bloated, hard-to-read, slow code
- **Redundant code is (mostly) a bad idea**
  - Design by contract makes code lean
  - Design by contract removes redundancy

Benefits of Design by Contract

- Leads to well-specified interfaces
- Leads to clean separation of work
- Makes software more reliable

Dilbert on Bugs

2. Expressing Contracts

Expressing Contracts

1. Preconditions
2. Class invariants
3. Postconditions

Preconditions

- **A boolean condition to be met for successful method entry**
  - The purpose is to guarantee a safe operating environment
  - If violated, the method should not be executed
- The client must make sure preconditions are met
  - A violation in the preconditions indicates a bug in the client
- Preconditions are method-level components of a contract

Preconditions of Name#insert(...)

```
public void insert(int i, String c) {
  assertClassInvariants();

  assertIsValidIndex(i, getNoComponents() + 1);
  assertIsNonNullArgument(c);
  int oldNoComponents = getNoComponents();

  doInsert(i, c);

  assert (oldNoComponents + 1) == getNoComponents() : "...";

  assertClassInvariants();
}

protected abstract void doInsert(int i, String c);
```

Postconditions

- **A boolean condition guaranteed after successful method exit**
  - If violated, the method failed to provide the service
- The method must make sure postconditions are met
  - A violation of a postcondition indicates a bug in the method
- Postconditions are method-level components of a contract

Postconditions of Name#insert(...)

```
public void insert(int i, String c) {
  assertClassInvariants();

  assertIsValidIndex(i, getNoComponents() + 1);
  assertIsNonNullArgument(c);
  int oldNoComponents = getNoComponents();

  doInsert(i, c);

  assert (oldNoComponents + 1) == getNoComponents() : "...";

  assertClassInvariants();
}

protected abstract void doInsert(int i, String c);
```

Class Invariants

- **A boolean condition that is true for any valid object**
  - Permanent violation of the class invariant indicates a broken object
  - Temporary violation is possible during method execution
- Class invariants are constraints on the object's state space
  - The class (implementation) must make sure its invariants are maintained
- Class invariants are class-level components of a contract

Class Invariants of Name

```
public void insert(int i, String c) {
  assertClassInvariants();

  assertIsValidIndex(i, getNoComponents() + 1);
  assertIsNonNullArgument(c);
  int oldNoComponents = getNoComponents();

  doInsert(i, c);

  assert (oldNoComponents + 1) == getNoComponents() : "...";

  assertClassInvariants();
}

protected abstract void doInsert(int i, String c);

protected void assertClassInvariants() {
  assert getNoComponents() >= 0;
  ...
}
```

3. Realizing Contracts

Realizing Design by Contract

- Annotate interface with class invariants
- Annotate methods with pre- and postconditions

Implementing Design by Contract

- In Java, use **`assert`** or assertion methods
  - Preconditions guard the entry to the method
  - Postconditions ensure successful completion
  - Class invariants add to postconditions

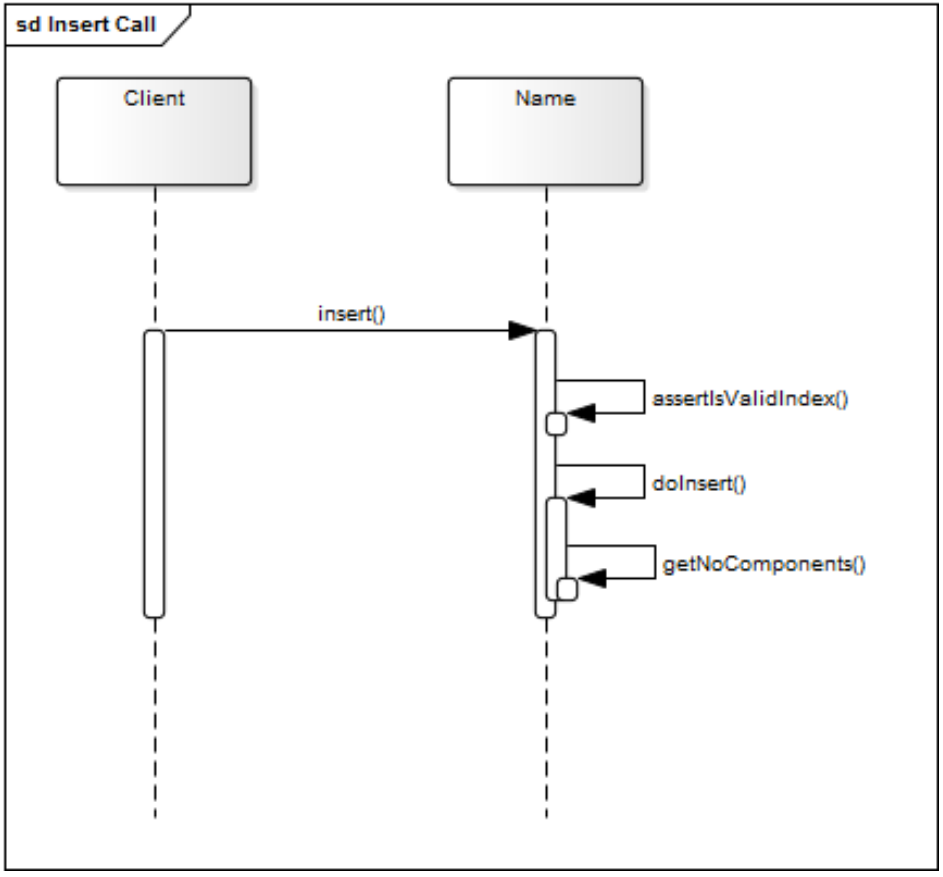
- **Assertions should be side-effect free**
  - Do not call mutation methods

Use of Assertion Methods

- Use dedicated assertion method if …
  - The assertion is used more than twice
  - A generic assertion failure exception is insufficient
  - You want to avoid that your assertion checking can be switched off

```
protected void assertIsValidIndex(int i, int upperLimit) throws IndexOutOfBoundsException {
  if ((i < 0) || (i >= upperLimit)) {
    String msg = String.valueOf(i) + "(of " + String.valueOf(getNoComponents()) +")";
    throw new IndexOutOfBoundsException(msg);
  }
}
```

Design by Contract and Assertions



Semantics of Various Exceptions

- IllegalArgumentException
- IndexOutOfBoundsException
- NullPointerException
- IllegalStateException
- ...

4. Contract Violation

Normal vs. Abnormal Operation

- **Normal situation**
  - “All is good” about the program; system is performing its function
  - Program pointer is in regular code, not exception handling code
- **Abnormal situation**
  - A contract was violated; the system needs to recover from the violation
  - Violation is detected by failing pre- and postconditions, class invariants
- **Modern programming languages distinguish both modes**
  - Contract violation leads to exceptions being thrown
  - Exception handling takes place outside regular code

Contracts and Control Flow [1]

	Control Flow	
	Normal Return by `return`	Abnormal Return by Exception
Contract Fulfilled	x	–
Contract Not Fulfilled	–	x

[1] Meyer's first two "laws" of exception handling reframed.

Handling Contract Violations

Handling an Exception

- Resumption
- Organized Panic

More on this in lecture on error handling

Two Alternatives of Handling Violations

- Resumption
  - Try again
  - Try alternative implementation
- Organized panic
  - Clean up as far as possible
  - Repackage exception, pass on

Quiz: Switching off Assertions

- Should you switch off assertions? (If so, when?)
  - Yes
  - No

5. Contracts in Context

Pragmatics of Design by Contract

- Use design by contract ...
  - For code that needs to be highly reliable
  - For code that is called often
- **Focus on preconditions to ...**
  - **Protect method operations**
  - **Document expectations**
- Why not postconditions and class invariants?
  - One method’s postconditions are another method’s precondition
  - Class invariants are implied postconditions for all methods
  - Pragmatically, tests already check for contract fulfillment

Design by Contract and Multithreading

```
public void insert(int i, String c) {
  assertIsValidIndex(i, getNoComponents() + 1);
  assertIsNonNullArgument(c);
  int oldNoComponents = getNoComponents();
  doInsert(i, c);
  assert (oldNoComponents + 1) == getNoComponents() : "...";
}
```

```
protected void doInsert(int index, String component) {
  int newSize = getNoComponents() + 1;
  String[] newComponents = new String[newSize];
  for (int i = 0, j = 0; j < newSize; j++) {
    if (j != index) {
      newComponents[j] = components[i++];
    } else {
      newComponents[j] = component;
    }
  }
  components = newComponents;
}
```

Design by Contract and Inheritance

- Preconditions may “accept more cases”
  - Subclasses may require less (weaken preconditions)
    - Preconditions get disjunctively (“or”) connected
  - Subclasses may contravariantly redefine method argument types
- Postconditions may “provide better results”
  - Subclasses may provide more (strengthen postconditions)
    - Postconditions get conjunctively (“and”) connected
  - Subclasses may covariantly redefine return type
- Class invariants may “provide better results”
  - Subclass invariants may not require less
  - If they provide more, they must refine superclass invariants

Preconditions and super.method() Calls

- Ensure preconditions of super.method(...) before using it

```
public Name AbstractName#insert(int index, String component) {
  assertIsValidIndex(index, getNoComponents() + 1);
  return doInsert(index, component);
}
```

```
public LazyName LazyName#insert(int index, String component) {
  ensureLength(index);
  return super.insert(index, componenht);
}

protected void ensureLength(int length) {
  // extend internal representation with empty components
  ...
}
```

Postconditions and Dual Hierarchies

- Covariant redefinition of return type strengthens postcondition

```
public Name AbstractName#insert(int index, String component) {
  assertIsValidIndex(index, getNoComponents() + 1);
  return doInsert(index, component);
}
```

```
public LazyName LazyName#insert(int index, String component) {
  ensureLength(index);
  return super.insert(index, component);
}

protected void ensureLength(int length) {
  // extend internal representation with empty components
  ...
}
```

Quiz: Violating Class Invariants

- Can you violate a class invariant in between two methods calls?
  - Yes
  - No

Summary

1. Design by contract
2. Expressing contracts
3. Realizing contracts
4. Contract violations
5. Contracts in context

Thank you! Questions?

dirk.riehle@fau.de – https://oss.cs.fau.de

dirk@riehle.org – https://dirkriehle.com – @dirkriehle

Legal Notices

- License
  - Licensed under the CC BY 4.0 International License
- Copyright
  - © 2012-2022 Dirk Riehle, some rights reserved