

Design-by-Contract (DbC)

Readings: OOSC2 Chapter 11



EECS3311 M: Software Design
Winter 2019

CHEN-WEI WANG

Motivation: Catching Defects – Design or Implementation Phase?



- To minimize **development costs**, minimize **software defects**.
∴ The cost of fixing defects **increases exponentially** as software progresses through the development lifecycle:
Requirements → **Design** → **Implementation** → Release
∴ Catch defects **as early as possible**.

Design and architecture	Implementation	Integration testing	Customer beta test	Postproduct release
1X*	5X	10X	15X	30X

- Discovering **defects** after **release** costs up to **30 times more** than catching them in the **design** phase.
- Choice of **design language** for your project is therefore of paramount importance.

Source: Minimizing code defects to improve software quality and lower development costs.
2 of 58

What This Course Is About



- Focus is **design**
 - Architecture**: (many) **inter-related** modules
 - Specification**: **precise** (functional) interface of each module
- For this course, having a prototypical, **working** implementation for your design suffices.
- A later **refinement** into more efficient data structures and algorithms is beyond the scope of this course.
[assumed from EECS2011, EECS3101]
- ∴ Having a suitable language for **design** matters the most.
- Q**: Is Java also a “good” **design** language?
- A**: Let’s first understand what a “good” **design** is.

3 of 58

Terminology: Contract, Client, Supplier



- A **supplier** implements/provides a service (e.g., microwave).
- A **client** uses a service provided by some supplier.
 - The client are required to follow certain instructions to obtain the service (e.g., supplier **assumes** that client powers on, closes door, and heats something that is not explosive).
 - If instructions are followed, the client would **expect** that the service does **what** is guaranteed (e.g., a lunch box is heated).
 - The client does not care **how** the supplier implements it.
- What then are the **benefits** and **obligations** as the two parties?

	benefits	obligations
CLIENT	obtain a service	follow instructions
SUPPLIER	assume instructions followed	provide a service

- There is a **contract** between two parties, **violated** if:
 - The instructions are not followed. [Client’s fault]
 - Instructions followed, but service not satisfactory. [Supplier’s fault]

4 of 58

Client, Supplier, Contract in OOP (1)



```
class Microwave {
    private boolean on;
    private boolean locked;
    void power() {on = true;}
    void lock() {locked = true;}
    void heat(Object stuff) {
        /* Assume: on && locked */
        /* stuff not explosive. */
    }
}
```

```
class MicrowaveUser {
    public static void main(...) {
        Microwave m = new Microwave();
        Object obj = ???;
        m.power(); m.lock();
        m.heat(obj);
    }
}
```

Method call `m.heat(obj)` indicates a client-supplier relation.

- **Client:** resident class of the method call [MicrowaveUser]
- **Supplier:** type of context object (or call target) `m` [Microwave]

5 of 58

What is a Good Design?



- A “good” design should *explicitly* and *unambiguously* describe the **contract** between **clients** (e.g., users of Java classes) and **suppliers** (e.g., developers of Java classes). We such a contractual relation a **specification**.
- When you conduct *software design*, you should be guided by the “appropriate” contracts between users and developers.
 - Instructions to **clients** should *not be unreasonable*.
e.g., asking them to assemble internal parts of a microwave
 - Working conditions for **suppliers** should *not be unconditional*.
e.g., expecting them to produce a microwave which can safely heat an explosive with its door open!
 - You as a designer should strike proper balance between **obligations** and **benefits** of clients and suppliers.
e.g., What is the obligation of a binary-search user (also benefit of a binary-search implementer)? [The input array is *sorted*.]
 - Upon contract violation, there should be the fault of **only one side**.
 - This design process is called **Design by Contract (DbC)**.

7 of 58

Client, Supplier, Contract in OOP (2)



```
class Microwave {
    private boolean on;
    private boolean locked;
    void power() {on = true;}
    void lock() {locked = true;}
    void heat(Object stuff) {
        /* Assume: on && locked */
        /* stuff not explosive. */
    }
}
```

```
class MicrowaveUser {
    public static void main(...) {
        Microwave m = new Microwave();
        Object obj = ???;
        m.power(); m.lock();
        m.heat(obj);
    }
}
```

- The **contract** is *honoured* if:

Right **before** the method call:

- State of `m` is as assumed: `m.on==true` and `m.locked==ture`
- The input argument `obj` is valid (i.e., not explosive).

Right **after** the method call: `obj` is properly heated.

- If any of these fails, there is a **contract violation**.
 - `m.on` or `m.locked` is false ⇒ MicrowaveUser's fault.
 - `obj` is an explosive ⇒ MicrowaveUser's fault.
 - A fault from the client is identified ⇒ Method call will not start.
 - Method executed but `obj` not properly heated ⇒ Microwave's fault

6 of 58

A Simple Problem: Bank Accounts



Provide an object-oriented solution to the following problem:

- REQ1**: Each account is associated with the *name* of its owner (e.g., "Jim") and an integer *balance* that is always positive.
- REQ2**: We may *withdraw* an integer amount from an account.
- REQ3**: Each bank stores a list of *accounts*.
- REQ4**: Given a bank, we may *add* a new account in it.
- REQ5**: Given a bank, we may *query* about the associated account of a owner (e.g., the account of "Jim").
- REQ6**: Given a bank, we may *withdraw* from a specific account, identified by its name, for an integer amount.

Let's first try to work on **REQ1** and **REQ2** in Java.
This may not be as easy as you might think!

8 of 58

Playing the Various Versions in Java



- **Download** the project archive (a zip file) here:
<http://www.eecs.yorku.ca/~jackie/teaching/lectures/2019/W/EECS3311/codes/DbCIntro.zip>
- Follow this tutorial to learn how to **import** an project archive into your workspace in Eclipse:
<https://youtu.be/h-rgdQZg2qY>
- Follow this tutorial to learn how to **enable** assertions in Eclipse:
<https://youtu.be/OEgRV4a5Dzg>

9 of 58

Version 1: Why Not a Good Design? (1)



```
public class BankAppV1 {  
    public static void main(String[] args) {  
        System.out.println("Create an account for Alan with balance -10:");  
        AccountV1 alan = new AccountV1("Alan", -10);  
        System.out.println(alan);  
    }  
}
```

Console Output:

```
Create an account for Alan with balance -10:  
Alan's current balance is: -10
```

- Executing AccountV1's constructor results in an account object whose **state** (i.e., values of attributes) is **invalid** (i.e., Alan's balance is negative). ⇒ Violation of **REQ1**
- Unfortunately, both client and supplier are to be blamed: BankAppV1 passed an invalid balance, but the API of AccountV1 does not require that! ⇒ A lack of defined contract

11 of 58

Version 1: An Account Class



```
1 public class AccountV1 {  
2     private String owner;  
3     private int balance;  
4     public String getOwner() { return owner; }  
5     public int getBalance() { return balance; }  
6     public AccountV1(String owner, int balance) {  
7         this.owner = owner; this.balance = balance;  
8     }  
9     public void withdraw(int amount) {  
10        this.balance = this.balance - amount;  
11    }  
12    public String toString() {  
13        return owner + "'s current balance is: " + balance;  
14    }  
15 }
```

- Is this a good design? Recall **REQ1**: Each account is associated with ... an integer balance that is **always positive**.
- This requirement is **not** reflected in the above Java code.

10 of 58

Version 1: Why Not a Good Design? (2)



```
public class BankAppV1 {  
    public static void main(String[] args) {  
        System.out.println("Create an account for Mark with balance 100:");  
        AccountV1 mark = new AccountV1("Mark", 100);  
        System.out.println(mark);  
        System.out.println("Withdraw -1000000 from Mark's account:");  
        mark.withdraw(-1000000);  
        System.out.println(mark);  
    }  
}
```

```
Create an account for Mark with balance 100:  
Mark's current balance is: 100  
Withdraw -1000000 from Mark's account:  
Mark's current balance is: 1000100
```

- Mark's account state is always valid (i.e., 100 and 1000100).
- Withdraw amount is never negative! ⇒ Violation of **REQ2**
- Again a lack of contract between BankAppV1 and AccountV1.

12 of 58

Version 1: Why Not a Good Design? (3)



```
public class BankAppV1 {
    public static void main(String[] args) {
        System.out.println("Create an account for Tom with balance 100:");
        AccountV1 tom = new AccountV1("Tom", 100);
        System.out.println(tom);
        System.out.println("Withdraw 150 from Tom's account:");
        tom.withdraw(150);
        System.out.println(tom);
    }
}
```

```
Create an account for Tom with balance 100:
Tom's current balance is: 100
Withdraw 150 from Tom's account:
Tom's current balance is: -50
```

- Withdrawal was done via an “appropriate” reduction, but the resulting balance of Tom is *invalid*. ⇒ Violation of **REQ1**
- Again a lack of contract between BankAppV1 and AccountV1.

13 of 58

Version 1: How Should We Improve it? (2)



The best we can do in Java is to encode the *logical negations* of preconditions as *exceptions*:

- divide(int x, int y)
throws DivisionByZeroException when $y == 0$.
- binSearch(int x, int[] xs)
throws ArrayNotSortedException when xs is *not* sorted.
- topoSort(Graph g)
throws NotDAGException when g is *not* directed and acyclic.
- Design your method by specifying the *preconditions* (i.e., *service* conditions for *valid* inputs) it requires, not the *exceptions* (i.e., *error* conditions for *invalid* inputs) for it to fail.
- Create **Version 2** by adding *exceptional conditions* (an *approximation* of *preconditions*) to the constructor and withdraw method of the Account class.

15 of 58

Version 1: How Should We Improve it? (1)



Preconditions of a method specify the precise circumstances under which that method can be executed.

- Precond. of divide(int x, int y)? [$y \neq 0$]
- Precond. of binSearch(int x, int[] xs)? [xs is sorted]
- Precond. of topoSort(Graph g)? [g is a DAG]

14 of 58

Version 2: Added Exceptions to Approximate Method Preconditions



```
1 public class AccountV2 {
2     public AccountV2(String owner, int balance) throws
3         BalanceNegativeException
4     {
5         if (balance < 0) { /* negated precondition */
6             throw new BalanceNegativeException(); }
7         else { this.owner = owner; this.balance = balance; }
8     }
9     public void withdraw(int amount) throws
10        WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
11         if (amount < 0) { /* negated precondition */
12             throw new WithdrawAmountNegativeException(); }
13         else if (balance < amount) { /* negated precondition */
14             throw new WithdrawAmountTooLargeException(); }
15         else { this.balance = this.balance - amount; }
16     }
}
```

16 of 58

Version 2: Why Better than Version 1? (1)



```
1 public class BankAppV2 {
2     public static void main(String[] args) {
3         System.out.println("Create an account for Alan with balance -10:");
4         try {
5             AccountV2 alan = new AccountV2("Alan", -10);
6             System.out.println(alan);
7         }
8         catch (BalanceNegativeException bne) {
9             System.out.println("Illegal negative account balance.");
10        }
11    }
12 }
```

```
Create an account for Alan with balance -10:
Illegal negative account balance.
```

L6: When attempting to call the constructor AccountV2 with a negative balance -10, a BalanceNegativeException (i.e., **precondition** violation) occurs, *preventing further operations upon this invalid object*.

17 of 58

Version 2: Why Better than Version 1? (2.2)



Console Output:

```
Create an account for Mark with balance 100:
Mark's current balance is: 100
Withdraw -1000000 from Mark's account:
Illegal negative withdraw amount.
```

- L9: When attempting to call method withdraw with a positive but too large amount 150, a WithdrawAmountTooLargeException (i.e., **precondition** violation) occurs, *preventing the withdrawal from proceeding*.
- We should observe that *adding preconditions* to the supplier BankV2's code forces the client BankAppV2's code to *get complicated by the try-catch statements*.
- Adding clear contract (**preconditions** in this case) to the design **should not** be at the cost of complicating the client's code!!

19 of 58

Version 2: Why Better than Version 1? (2.1)



```
1 public class BankAppV2 {
2     public static void main(String[] args) {
3         System.out.println("Create an account for Mark with balance 100:");
4         try {
5             AccountV2 mark = new AccountV2("Mark", 100);
6             System.out.println(mark);
7             System.out.println("Withdraw -1000000 from Mark's account:");
8             mark.withdraw(-1000000);
9             System.out.println(mark);
10        }
11        catch (BalanceNegativeException bne) {
12            System.out.println("Illegal negative account balance.");
13        }
14        catch (WithdrawAmountNegativeException wane) {
15            System.out.println("Illegal negative withdraw amount.");
16        }
17        catch (WithdrawAmountTooLargeException wane) {
18            System.out.println("Illegal too large withdraw amount.");
19        }
20    }
21 }
```

18 of 58

Version 2: Why Better than Version 1? (3.1)



```
1 public class BankAppV2 {
2     public static void main(String[] args) {
3         System.out.println("Create an account for Tom with balance 100:");
4         try {
5             AccountV2 tom = new AccountV2("Tom", 100);
6             System.out.println(tom);
7             System.out.println("Withdraw 150 from Tom's account:");
8             tom.withdraw(150);
9             System.out.println(tom);
10        }
11        catch (BalanceNegativeException bne) {
12            System.out.println("Illegal negative account balance.");
13        }
14        catch (WithdrawAmountNegativeException wane) {
15            System.out.println("Illegal negative withdraw amount.");
16        }
17        catch (WithdrawAmountTooLargeException wane) {
18            System.out.println("Illegal too large withdraw amount.");
19        }
20    }
21 }
```

20 of 58

Version 2: Why Better than Version 1? (3.2)



Console Output:

```
Create an account for Tom with balance 100:
Tom's current balance is: 100
Withdraw 150 from Tom's account:
Illegal too large withdraw amount.
```

- **L9**: When attempting to call method `withdraw` with a negative amount `-1000000`, a `WithdrawAmountNegativeException` (i.e., **precondition** violation) occurs, *preventing the withdrawal from proceeding*.
- We should observe that due to the *added preconditions* to the supplier `BankV2`'s code, the client `BankAppV2`'s code is forced to *repeat the long list of the try-catch statements*.
- Indeed, adding clear contract (*preconditions* in this case) **should not** be at the cost of complicating the client's code!!

21 of 58

Version 2: Why Still Not a Good Design? (2.1)



```
1 public class BankAppV2 {
2     public static void main(String[] args) {
3         System.out.println("Create an account for Jim with balance 100:");
4         try {
5             AccountV2 jim = new AccountV2("Jim", 100);
6             System.out.println(jim);
7             System.out.println("Withdraw 100 from Jim's account:");
8             jim.withdraw(100);
9             System.out.println(jim);
10        }
11        catch (BalanceNegativeException bne) {
12            System.out.println("Illegal negative account balance.");
13        }
14        catch (WithdrawAmountNegativeException wane) {
15            System.out.println("Illegal negative withdraw amount.");
16        }
17        catch (WithdrawAmountTooLargeException wane) {
18            System.out.println("Illegal too large withdraw amount.");
19        }
20    }
21 }
```

23 of 58

Version 2: Why Still Not a Good Design? (1)



```
1 public class AccountV2 {
2     public AccountV2(String owner, int balance) throws
3         BalanceNegativeException
4     {
5         if (balance < 0) { /* negated precondition */
6             throw new BalanceNegativeException(); }
7         else { this.owner = owner; this.balance = balance; }
8     }
9     public void withdraw(int amount) throws
10        WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
11        if (amount < 0) { /* negated precondition */
12            throw new WithdrawAmountNegativeException(); }
13        else if (balance < amount) { /* negated precondition */
14            throw new WithdrawAmountTooLargeException(); }
15        else { this.balance = this.balance - amount; }
16    }
17 }
```

- Are all the *exception* conditions (\neg *preconditions*) appropriate?
- What if `amount == balance` when calling `withdraw`?

22 of 58

Version 2: Why Still Not a Good Design? (2.2)



```
Create an account for Jim with balance 100:
Jim's current balance is: 100
Withdraw 100 from Jim's account:
Jim's current balance is: 0
```

L9: When attempting to call method `withdraw` with an amount `100` (i.e., equal to Jim's current balance) that would result in a **zero** balance (clearly a violation of **REQ1**), there should have been a *precondition* violation.

Supplier `AccountV2`'s *exception* condition `balance < amount` has a **missing case**:

- Calling `withdraw` with `amount == balance` will also result in an invalid account state (i.e., the resulting account balance is **zero**).
- \therefore **L13** of `AccountV2` should be `balance <= amount`.

24 of 58

Version 2: How Should We Improve it?



- **Even without** fixing this insufficient *precondition*, we could have avoided the above scenario by *checking at the end of each method that the resulting account is valid*.
 - ⇒ We consider the condition `this.balance > 0` as **invariant** throughout the lifetime of all instances of `Account`.
- **Invariants** of a class specify the precise conditions which all instances/objects of that class must satisfy.
 - Inv. of `CSMajorStudent`? [`gpa >= 4.5`]
 - Inv. of `BinarySearchTree`? [in-order trav. → sorted key seq.]
- The best we can do in Java is encode invariants as **assertions**:
 - `CSMajorStudent`: **assert** `this.gpa >= 4.5`
 - `BinarySearchTree`: **assert** `this.inOrder()` is sorted
 - Unlike exceptions, assertions are not in the class/method API.
- Create **Version 3** by adding **assertions** to the end of constructor and `withdraw` method of the `Account` class.

25 of 58

Version 3: Why Better than Version 2?



```
1 public class BankAppV3 {
2     public static void main(String[] args) {
3         System.out.println("Create an account for Jim with balance 100:");
4         try { AccountV3 jim = new AccountV3("Jim", 100);
5             System.out.println(jim);
6             System.out.println("Withdraw 100 from Jim's account:");
7             jim.withdraw(100);
8             System.out.println(jim); }
9         /* catch statements same as this previous slide:
10         * Version 2: Why Still Not a Good Design? (2.1) */
```

```
Create an account for Jim with balance 100:
Jim's current balance is: 100
Withdraw 100 from Jim's account:
Exception in thread "main"
```

java.lang.AssertionError: Invariant: positive balance

L8: Upon completion of `jim.withdraw(100)`, Jim has a zero balance, an assertion failure (i.e., **invariant** violation) occurs, *preventing further operations on this invalid account object.*

27 of 58

Version 3: Added Assertions to Approximate Class Invariants



```
1 public class AccountV3 {
2     public AccountV3(String owner, int balance) throws
3         BalanceNegativeException
4     {
5         if(balance < 0) { /* negated precondition */
6             throw new BalanceNegativeException(); }
7         else { this.owner = owner; this.balance = balance; }
8         assert this.getBalance() > 0 : "Invariant: positive balance";
9     }
10    public void withdraw(int amount) throws
11        WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
12        if(amount < 0) { /* negated precondition */
13            throw new WithdrawAmountNegativeException(); }
14        else if (balance < amount) { /* negated precondition */
15            throw new WithdrawAmountTooLargeException(); }
16        else { this.balance = this.balance - amount; }
17        assert this.getBalance() > 0 : "Invariant: positive balance";
18    }
```

26 of 58

Version 3: Why Still Not a Good Design?



Let's review what we have added to the method `withdraw`:

- From **Version 2**: **exceptions** encoding **negated preconditions**
- From **Version 3**: **assertions** encoding the **class invariants**

```
1 public class AccountV3 {
2     public void withdraw(int amount) throws
3         WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
4         if(amount < 0) { /* negated precondition */
5             throw new WithdrawAmountNegativeException(); }
6         else if (balance < amount) { /* negated precondition */
7             throw new WithdrawAmountTooLargeException(); }
8         else { this.balance = this.balance - amount; }
9         assert this.getBalance() > 0 : "Invariant: positive balance";
10    }
```

However, there is **no contract** in `withdraw` which specifies:

- Obligations of supplier (`AccountV3`) if preconditions are met.
 - Benefits of client (`BankAppV3`) after meeting preconditions.
- ⇒ We illustrate how problematic this can be by creating

Version 4, where deliberately mistakenly implement `withdraw`.

28 of 58

Version 4: What If the Implementation of withdraw is Wrong? (1)

```

1 public class AccountV4 {
2     public void withdraw(int amount) throws
3         WithdrawAmountNegativeException, WithdrawAmountTooLargeException
4     { if(amount < 0) { /* negated precondition */
5         throw new WithdrawAmountNegativeException(); }
6         else if (balance < amount) { /* negated precondition */
7             throw new WithdrawAmountTooLargeException(); }
8         else { /* WRONG IMPLEMENTATION */
9             this.balance = this.balance + amount; }
10        assert this.getBalance() > 0 :
11            owner + "Invariant: positive balance"; }

```

- Apparently the implementation at L11 is **wrong**.
- Adding a positive amount to a valid (positive) account balance would not result in an invalid (negative) one.
⇒ The **class invariant** will **not** catch this flaw.
- When something goes wrong, a good **design** (with an appropriate **contract**) should report it via a **contract violation**.

29 of 58

Version 4: How Should We Improve it?

- **Postconditions** of a method specify the precise conditions which it will satisfy upon its completion.
 - This relies on the assumption that right before the method starts, its preconditions are satisfied (i.e., inputs valid) and invariants are satisfied (i.e., object state valid).
 - Postcondition of double divide(int x, int y)?
[**Result** × y == x]
 - Postcondition of boolean binSearch(int x, int[] xs)?
[$x \in XS \iff$ **Result**]
- The best we can do in Java is, similar to the case of invariants, encode postconditions as **assertions**.
But again, unlike exceptions, these assertions will not be part of the class/method API.
- Create **Version 5** by adding **assertions** to the end of withdraw method of the Account class.

31 of 58

Version 4: What If the Implementation of withdraw is Wrong? (2)

```

1 public class BankAppV4 {
2     public static void main(String[] args) {
3         System.out.println("Create an account for Jeremy with balance 100:");
4         try { AccountV4 jeremy = new AccountV4("Jeremy", 100);
5             System.out.println(jeremy);
6             System.out.println("Withdraw 50 from Jeremy's account:");
7             jeremy.withdraw(50);
8             System.out.println(jeremy); }
9         /* catch statements same as this previous slide:
10        * Version 2: Why Still Not a Good Design? (2.1) */

```

```

Create an account for Jeremy with balance 100:
Jeremy's current balance is: 100
Withdraw 50 from Jeremy's account:
Jeremy's current balance is: 150

```

L7: Resulting balance of Jeremy is valid (150 > 0), but withdrawal was done via an **mistaken** increase. ⇒ Violation of **REQ2**

30 of 58

Version 5: Added Assertions to Approximate Method Postconditions

```

1 public class AccountV5 {
2     public void withdraw(int amount) throws
3         WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
4         int oldBalance = this.balance;
5         if(amount < 0) { /* negated precondition */
6             throw new WithdrawAmountNegativeException(); }
7         else if (balance < amount) { /* negated precondition */
8             throw new WithdrawAmountTooLargeException(); }
9         else { this.balance = this.balance - amount; }
10        assert this.getBalance() > 0 : "Invariant: positive balance";
11        assert this.getBalance() == oldBalance - amount :
12            "Postcondition: balance deducted"; }

```

A postcondition typically relates the pre-execution value and the post-execution value of each relevant attribute (e.g., balance in the case of withdraw).
⇒ Extra code (L4) to capture the pre-execution value of balance for the comparison at L11.

32 of 58

Version 5: Why Better than Version 4?



```

1 public class BankAppV5 {
2     public static void main(String[] args) {
3         System.out.println("Create an account for Jeremy with balance 100:");
4         try { AccountV5 jeremy = new AccountV5("Jeremy", 100);
5             System.out.println(jeremy);
6             System.out.println("Withdraw 50 from Jeremy's account:");
7             jeremy.withdraw(50);
8             System.out.println(jeremy); }
9         /* catch statements same as this previous slide:
10        * Version 2: Why Still Not a Good Design? (2.1) */
    
```

```

Create an account for Jeremy with balance 100:
Jeremy's current balance is: 100
Withdraw 50 from Jeremy's account:
Exception in thread "main"
    java.lang.AssertionError: Postcondition: balance deducted
    
```

L8: Upon completion of `jeremy.withdraw(50)`, Jeremy has a wrong balance 150, an assertion failure (i.e., **postcondition** violation) occurs, *preventing further operations on this invalid account object*.

33 of 58

Version 5: Contract between Client and Supplier



	benefits	obligations
BankAppV5.main (CLIENT)	balance deduction positive balance	amount non-negative amount not too large
BankV5.withdraw (SUPPLIER)	amount non-negative amount not too large	balance deduction positive balance

	benefits	obligations
CLIENT	postcondition & invariant	precondition
SUPPLIER	precondition	postcondition & invariant

35 of 58

Evolving from Version 1 to Version 5



	Improvements Made	Design Flaws
V1	–	Complete lack of Contract
V2	Added exceptions as method preconditions	Preconditions not strong enough (i.e., with missing cases) may result in an invalid account state.
V3	Added assertions as class invariants	Incorrect implementations do not necessarily result in a state that violates the class invariants.
V4	Deliberately changed <code>withdraw</code> 's implementation to be incorrect .	The incorrect implementation does not result in a state that violates the class invariants.
V5	Added assertions as method postconditions	–

- In Versions 2, 3, 4, 5, **preconditions** approximated as **exceptions**.
 - ⊗ These are **not preconditions**, but their **logical negation**.
 - ⊗ Client `BankApp`'s code **complicated** by repeating the list of `try-catch` statements.
- In Versions 3, 4, 5, **class invariants** and **postconditions** approximated as **assertions**.
 - ⊗ Unlike exceptions, these assertions will **not appear in the API** of `withdraw`. Potential clients of this method **cannot know**: 1) what their benefits are; and 2) what their suppliers' obligations are.
 - ⊗ For postconditions, **extra code** needed to capture pre-execution values of attributes.

34 of 58

DbC in Java



DbC is possible in Java, but not appropriate for your learning:

- Preconditions** of a method:
 - Supplier**
 - Encode their logical negations as exceptions.
 - In the **beginning** of that method, a list of `if`-statements for throwing the appropriate exceptions.
 - Client**
 - A list of `try-catch`-statements for handling exceptions.
- Postconditions** of a method:
 - Supplier**
 - Encoded as a list of assertions, placed at the **end** of that method.
 - Client**
 - All such assertions do not appear in the API of that method.
- Invariants** of a class:
 - Supplier**
 - Encoded as a list of assertions, placed at the **end** of **every** method.
 - Client**
 - All such assertions do not appear in the API of that class.

36 of 58

DbC in Eiffel: Supplier

DbC is supported natively in Eiffel for **supplier**:

```
class ACCOUNT
create
  make
feature -- Attributes
  owner : STRING
  balance : INTEGER
feature -- Constructors
  make(nn: STRING; nb: INTEGER)
    require -- precondition
      positive_balance: nb > 0
    do
      owner := nn
      balance := nb
    end
feature -- Commands
  withdraw(amount: INTEGER)
    require -- precondition
      non_negative_amount: amount > 0
      affordable_amount: amount <= balance -- problematic, why?
    do
      balance := balance - amount
    ensure -- postcondition
      balance_deducted: balance = old balance - amount
    end
invariant -- class invariant
  positive_balance: balance > 0
end
37 of 58
```

DbC in Eiffel: Anatomy of a Class

```
class SOME_CLASS
create
  -- Explicitly list here commands used as constructors
feature -- Attributes
  -- Declare attribute here
feature -- Commands
  -- Declare commands (mutators) here
feature -- Queries
  -- Declare queries (accessors) here
invariant
  -- List of tagged boolean expressions for class invariants
end
```

- Use **feature** clauses to group attributes, commands, queries.
- Explicitly declare list of commands under **create** clause, so that they can be used as class constructors. [See the groups panel in Eiffel Studio.]
- The **class invariant invariant** clause may be omitted:
 - There's no class invariant: any resulting object state is acceptable.
 - The class invariant is equivalent to writing **invariant true**

39 of 58

DbC in Eiffel: Contract View of Supplier

Any potential **client** who is interested in learning about the kind of services provided by a **supplier** can look through the **contract view** (without showing any implementation details):

```
class ACCOUNT
create
  make
feature -- Attributes
  owner : STRING
  balance : INTEGER
feature -- Constructors
  make(nn: STRING; nb: INTEGER)
    require -- precondition
      positive_balance: nb > 0
    end
feature -- Commands
  withdraw(amount: INTEGER)
    require -- precondition
      non_negative_amount: amount > 0
      affordable_amount: amount <= balance -- problematic, why?
    ensure -- postcondition
      balance_deducted: balance = old balance - amount
    end
invariant -- class invariant
  positive_balance: balance > 0
end
```

38 of 58

DbC in Eiffel: Anatomy of a Feature

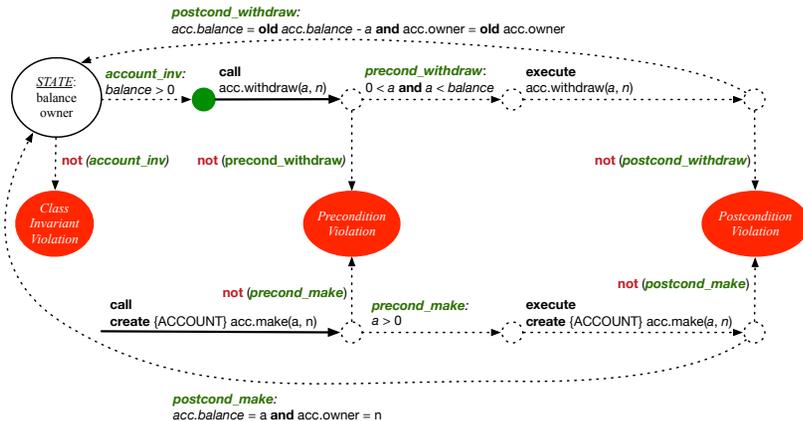
```
some_command
  -- Description of the command.
  require
    -- List of tagged boolean expressions for preconditions
  local
    -- List of local variable declarations
  do
    -- List of instructions as implementation
  ensure
    -- List of tagged boolean expressions for postconditions
  end
```

- The **precondition require** clause may be omitted:
 - There's no precondition: any starting state is acceptable.
 - The precondition is equivalent to writing **require true**
- The **postcondition ensure** clause may be omitted:
 - There's no postcondition: any resulting state is acceptable.
 - The postcondition is equivalent to writing **ensure true**

40 of 58

Runtime Monitoring of Contracts (1)

In the specific case of ACCOUNT class with creation procedure `make` and command `withdraw`:



41 of 58

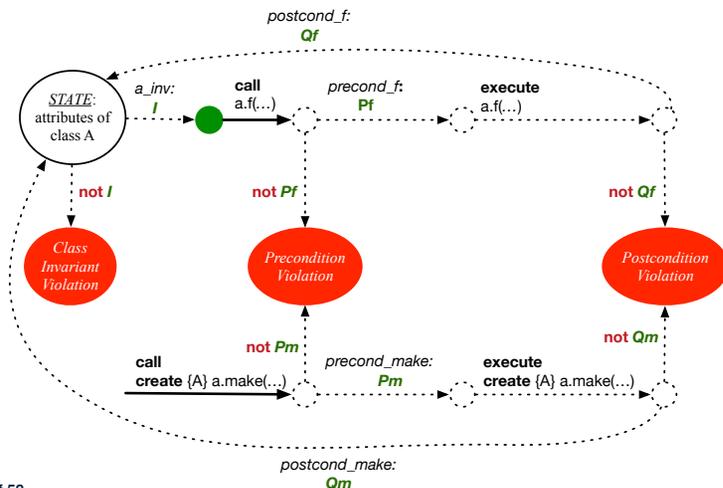
Runtime Monitoring of Contracts (3)

- All **contracts** are specified as **Boolean expressions**.
- Right **before** a feature call (e.g., `acc.withdraw(10)`):
 - The current state of `acc` is called the **pre-state**.
 - Evaluate feature `withdraw`'s **pre-condition** using current values of attributes and queries.
 - Cache** values (**implicitly**) of all expressions involving the **old** keyword in the **post-condition**.
e.g., cache the value of `old balance` via `old_balance := balance`
- Right **after** the feature call:
 - The current state of `acc` is called the **post-state**.
 - Evaluate class ACCOUNT's **invariant** using current values of attributes and queries.
 - Evaluate feature `withdraw`'s **post-condition** using both current and **"cached"** values of attributes and queries.

43 of 58

Runtime Monitoring of Contracts (2)

In general, class C with creation procedure `cp` and any feature `f`:



42 of 58

DbC in Eiffel: Precondition Violation (1.1)

The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit
  ARGUMENTS
create
  make
feature -- Initialization
  make
  -- Run application.
  local
    alan: ACCOUNT
  do
    -- A precondition violation with tag "positive_balance"
    create {ACCOUNT} alan.make ("Alan", -10)
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a **contract violation** (precondition violation with tag "positive_balance").

44 of 58

DbC in Eiffel: Precondition Violation (1.2)



```
class ACCOUNT
  make (nn: STRING_8; nb: INTEGER_32)
  require
    positive_balance: nb >= 0
  do
    owner := nn
    balance := nb
  end
```

45 of 58

DbC in Eiffel: Precondition Violation (2.2)



```
class ACCOUNT
  withdraw (amount: INTEGER_32)
  require
    non_negative_amount: amount >= 0
    affordable_amount: amount <= balance
  do
    balance := balance - amount
  ensure
    balance = old balance - amount
  end
```

47 of 58

DbC in Eiffel: Precondition Violation (2.1)

The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit
  ARGUMENTS
create
  make
feature -- Initialization
  make
  -- Run application.
  local
    mark: ACCOUNT
  do
    create {ACCOUNT} mark.make ("Mark", 100)
    -- A precondition violation with tag "non_negative_amount"
    mark.withdraw(-1000000)
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a **contract violation** (precondition violation with tag "non_negative_amount").

46 of 58

DbC in Eiffel: Precondition Violation (3.1)

The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit
  ARGUMENTS
create
  make
feature -- Initialization
  make
  -- Run application.
  local
    tom: ACCOUNT
  do
    create {ACCOUNT} tom.make ("Tom", 100)
    -- A precondition violation with tag "affordable_amount"
    tom.withdraw(150)
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a **contract violation** (precondition violation with tag "affordable_amount").

48 of 58

DbC in Eiffel: Precondition Violation (3.2)



```
APPLICATION: ACCOUNT
Feature: bank ACCOUNT withdraw
Flat view of feature 'withdraw' of class ACCOUNT
withdraw (amount: INTEGER_32)
  require
    non_negative_amount: amount >= 0
    affordable_amount: amount <= balance
  do
    balance := balance - amount
  ensure
    balance = old balance - amount
  end
```

Call Stack: Status = Implicit exception pending
affordable_amount: PRECONDITION_VIOLATION raised

In Feature	In Class	From Class	@
withdraw	ACCOUNT	ACCOUNT	2
make	APPLICATION	APPLICATION	2

49 of 58

DbC in Eiffel: Class Invariant Violation (4.2)



```
APPLICATION: ACCOUNT
Feature: bank ACCOUNT _invariant
Flat view of feature '_invariant' of class ACCOUNT
_invariant
  positive_balance: balance > 0
```

Call Stack: Status = Implicit exception pending
positive_balance: INVARIANT_VIOLATION raised

In Feature	In Class	From Class	@
_invariant	ACCOUNT	ACCOUNT	0
withdraw	ACCOUNT	ACCOUNT	5
make	APPLICATION	APPLICATION	2

51 of 58

DbC in Eiffel: Class Invariant Violation (4.1)



The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit
  ARGUMENTS
create
  make
feature -- Initialization
  make
  -- Run application.
  local
    jim: ACCOUNT
  do
    create {ACCOUNT} tom.make ("Jim", 100)
    jim.withdraw(100)
    -- A class invariant violation with tag "positive_balance"
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a **contract violation** (class invariant violation with tag "positive_balance").

50 of 58

DbC in Eiffel: Postcondition Violation (5.1)



The **client** need not handle all possible contract violations:

```
class BANK_APP
inherit ARGUMENTS
create make
feature -- Initialization
  make
  -- Run application.
  local
    jeremy: ACCOUNT
  do
    -- Faulty implementation of withdraw in ACCOUNT:
    -- balance := balance + amount
    create {ACCOUNT} jeremy.make ("Jeremy", 100)
    jeremy.withdraw(150)
    -- A postcondition violation with tag "balance_deducted"
  end
end
```

By executing the above code, the runtime monitor of Eiffel Studio will report a **contract violation** (postcondition violation with tag "balance_deducted").

52 of 58

DbC in Eiffel: Postcondition Violation (5.2)



```
Feature
Flat view of feature 'withdraw' of class ACCOUNT
affordable_amount: amount <= balance
do
  balance := balance + amount
ensure
  (balance_deducted: balance = old balance - amount)
end
```

Call Stack
Status = Implicit exception pending
balance_deducted: POSTCONDITION_VIOLATION raised

In Feature	In Class	From Class	@
withdraw	ACCOUNT	ACCOUNT	4
make	APPLICATION	APPLICATION 2	

53 of 58

Index (1)



Motivation: Catching Defects –
Design or Implementation Phase?

What This Course Is About

Terminology: Contract, Client, Supplier

Client, Supplier, Contract in OOP (1)

Client, Supplier, Contract in OOP (2)

What is a Good Design?

A Simple Problem: Bank Accounts

Playing with the Various Versions in Java

Version 1: An Account Class

Version 1: Why Not a Good Design? (1)

Version 1: Why Not a Good Design? (2)

Version 1: Why Not a Good Design? (3)

Version 1: How Should We Improve it? (1)

54 of 58

Index (2)



Version 1: How Should We Improve it? (2)

Version 2: Added Exceptions
to Approximate Method Preconditions

Version 2: Why Better than Version 1? (1)

Version 2: Why Better than Version 1? (2.1)

Version 2: Why Better than Version 1? (2.2)

Version 2: Why Better than Version 1? (3.1)

Version 2: Why Better than Version 1? (3.2)

Version 2: Why Still Not a Good Design? (1)

Version 2: Why Still Not a Good Design? (2.1)

Version 2: Why Still Not a Good Design? (2.2)

Version 2: How Should We Improve it?

Version 3: Added Assertions

to Approximate Class Invariants

55 of 58

Index (3)



Version 3: Why Better than Version 2?

Version 3: Why Still Not a Good Design?

Version 4: What If the

Implementation of `withdraw` is Wrong? (1)

Version 4: What If the

Implementation of `withdraw` is Wrong? (2)

Version 4: How Should We Improve it?

Version 5: Added Assertions

to Approximate Method Postconditions

Version 5: Why Better than Version 4?

Evolving from Version 1 to Version 5

Version 5:

Contract between Client and Supplier

DbC in Java

DbC in Eiffel: Supplier

56 of 58

Index (4)



DbC in Eiffel: Contract View of Supplier
DbC in Eiffel: Anatomy of a Class
DbC in Eiffel: Anatomy of a Feature
Runtime Monitoring of Contracts (1)
Runtime Monitoring of Contracts (2)
Runtime Monitoring of Contracts (3)
DbC in Eiffel: Precondition Violation (1.1)
DbC in Eiffel: Precondition Violation (1.2)
DbC in Eiffel: Precondition Violation (2.1)
DbC in Eiffel: Precondition Violation (2.2)
DbC in Eiffel: Precondition Violation (3.1)
DbC in Eiffel: Precondition Violation (3.2)
DbC in Eiffel: Class Invariant Violation (4.1)
DbC in Eiffel: Class Invariant Violation (4.2)

57 of 58

Index (5)



DbC in Eiffel: Postcondition Violation (5.1)

DbC in Eiffel: Postcondition Violation (5.2)

58 of 58