Design-by-Contract (Dbc)
Test-Driven Development (TDD)

Readings: OOSC2 Chapter 11

EECS3311 A: Software Design
Fall 2018

CHEN-WEI WANG

Motivation of this Course

- Focus is *design*
  - ***Architecture***: work with many ***interacting*** classes
  - ***Specification***: being mathematically ***precise*** about expectations
- 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.

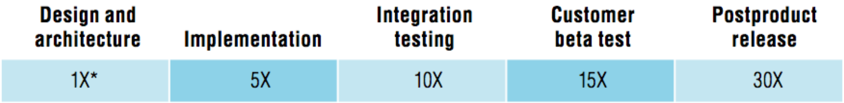
Catching Defects: Design or Implementation Phase?

- 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.

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* os 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]

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`]

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

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)*.

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!

Playing the Various Versions in Java

- **Download** the project archive (a zip file) here: `http://www.eecs.yorku.ca/~jackie/teaching/lectures/2018/F/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`

Version 1: An Account Class

```java
public class AccountV1 {
      private String owner;
      private int balance;
      public String getOwner() { return owner; }
      public int getBalance() { return balance; }
      public AccountV1(String owner, int balance) {
            this.owner = owner; this.balance = balance;
      }
      public void withdraw(int amount) {
            this.balance = this.balance - amount;
      }
      public String toString() {
            return owner + "'s current balance is: " + balance;
      }
}
```

- 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.

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

```java
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

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 $\boxed{\textbf{REQ2}}$
- Again a lack of contract between `BankAppV1` and `AccountV1`.

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`.

Version 1: How Should We Improve it?

- *Preconditions* of a method specify the precise circumstances under which that method can be executed.
  - Precond. of `divide(int x, int y)`? [`y != 0`]
  - Precond. of `binSearch(int x, int[] xs)`? [`xs` is sorted]
- 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.
  - It should be preferred to design your method by specifying the *preconditions* (i.e., *valid* inputs) it requires, rather than the *exceptions* (i.e., *erroneous* inputs) that it might trigger.
- Create Version 2 by adding *exceptional conditions* (an *approximation* of *preconditions*) to the constructor and `withdraw` method of the `Account` class.

Version 2: Added Exceptions to Approximate Method Preconditions

```
public class AccountV2 {
  public AccountV2(String owner, int balance) throws
      BalanceNegativeException
  {
    if( balance < 0 ) { /* negated precondition */
      throw new BalanceNegativeException(); }
    else { this.owner = owner; this.balance = balance; }
  }
  public void withdraw(int amount) throws
      WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
    if( amount < 0 ) { /* negated precondition */
      throw new WithdrawAmountNegativeException(); }
    else if ( balance < amount ) { /* negated precondition */
      throw new WithdrawAmountTooLargeException(); }
    else { this.balance = this.balance - amount; }
  }
```

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      }
```

```
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*.

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

```java
public class BankAppV2 {
  public static void main(String[] args) {
    System.out.println("Create an account for Mark with balance 100:");
    try {
      AccountV2 mark = new AccountV2("Mark", 100);
      System.out.println(mark);
      System.out.println("Withdraw -1000000 from Mark's account:");
      mark.withdraw(-1000000);
      System.out.println(mark);
    }
    catch (BalanceNegativeException bne) {
      System.out.println("Illegal negative account balance.");
    }
    catch (WithdrawAmountNegativeException wane) {
      System.out.println("Illegal negative withdraw amount.");
    }
    catch (WithdrawAmountTooLargeException wane) {
      System.out.println("Illegal too large withdraw amount.");
    }
```

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!!

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

```
public class BankAppV2 {
  public static void main(String[] args) {
    System.out.println("Create an account for Tom with balance 100:");
    try {
      AccountV2 tom = new AccountV2("Tom", 100);
      System.out.println(tom);
      System.out.println("Withdraw 150 from Tom's account:");
      tom.withdraw(150);
      System.out.println(tom);
    }
    catch (BalanceNegativeException bne) {
      System.out.println("Illegal negative account balance.");
    }
    catch (WithdrawAmountNegativeException wane) {
      System.out.println("Illegal negative withdraw amount.");
    }
    catch (WithdrawAmountTooLargeException wane) {
      System.out.println("Illegal too large withdraw amount.");
    }
```

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!!

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

```java
public class AccountV2 {
  public AccountV2(String owner, int balance) throws
      BalanceNegativeException
  {
    if( balance < 0 ) { /* negated precondition */
      throw new BalanceNegativeException(); }
    else { this.owner = owner; this.balance = balance; }
  }
  public void withdraw(int amount) throws
      WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
    if( amount < 0 ) { /* negated precondition */
      throw new WithdrawAmountNegativeException(); }
    else if ( balance < amount ) { /* negated precondition */
      throw new WithdrawAmountTooLargeException(); }
    else { this.balance = this.balance - amount; }
  }
```

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

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

```
public class BankAppV2 {
  public static void main(String[] args) {
    System.out.println("Create an account for Jim with balance 100:");
    try {
      AccountV2 jim = new AccountV2("Jim", 100);
      System.out.println(jim);
      System.out.println("Withdraw 100 from Jim’s account:");
      jim. withdraw(100) ;
      System.out.println(jim);
    }
    catch (BalanceNegativeException bne) {
      System.out.println("Illegal negative account balance.");
    }
    catch (WithdrawAmountNegativeException wane) {
      System.out.println("Illegal negative withdraw amount.");
    }
    catch (WithdrawAmountTooLargeException wane) {
      System.out.println("Illegal too large withdraw amount.");
    }
```

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**).
- ∴ **L13** of `AccountV2` should be `balance <= amount`.

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 `CSMajoarStudent`? [`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.

Version 3: Added Assertions to Approximate Class Invariants

```
public class AccountV3 {
  public AccountV3(String owner, int balance) throws
      BalanceNegativeException
  {
    if(balance < 0) { /* negated precondition */
      throw new BalanceNegativeException(); }
    else { this.owner = owner; this.balance = balance; }
    assert this.getBalance() > 0 : "Invariant:  positive balance";
  }
  public void withdraw(int amount) throws
      WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
    if(amount < 0) { /* negated precondition */
      throw new WithdrawAmountNegativeException(); }
    else if (balance < amount) { /* negated precondition */
      throw new WithdrawAmountTooLargeException(); }
    else { this.balance = this.balance - amount; }
    assert this.getBalance() > 0 : "Invariant:  positive balance";
  }
```

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*.

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

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*

```
public class AccountV3 {
  public void withdraw(int amount) throws
     WithdrawAmountNegativeException, WithdrawAmountTooLargeException {
    if( amount < 0 ) { /* negated precondition */
     throw new WithdrawAmountNegativeException(); }
    else if ( balance < amount ) { /* negated precondition */
     throw new WithdrawAmountTooLargeException(); }
    else { this.balance = this.balance - amount; }
    assert this.getBalance() > 0 : "Invariant: positive balance"; }
```

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`.

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 { /* WRONT 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*.

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**

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)`?

    $[\ \textbf{Result} \times y == x\ ]$

  - Postcondition of `boolean binSearch(int x, int[] xs)`?

    $[\ x \in xs \iff \textbf{Result}\ ]$

- 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 $\boxed{\text{Version 5}}$ by adding *assertions* to the end of `withdraw` method of the `Account` class.

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**.

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*.

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 `withdraw`'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.

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

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.

Index (1)

Index (2)

Index (3)