UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

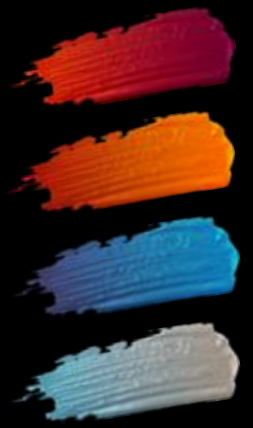
SERLAB
Software Engineering Research

Integrazione e Test di Sistemi Software

Designing contracts

Azzurra Ragone

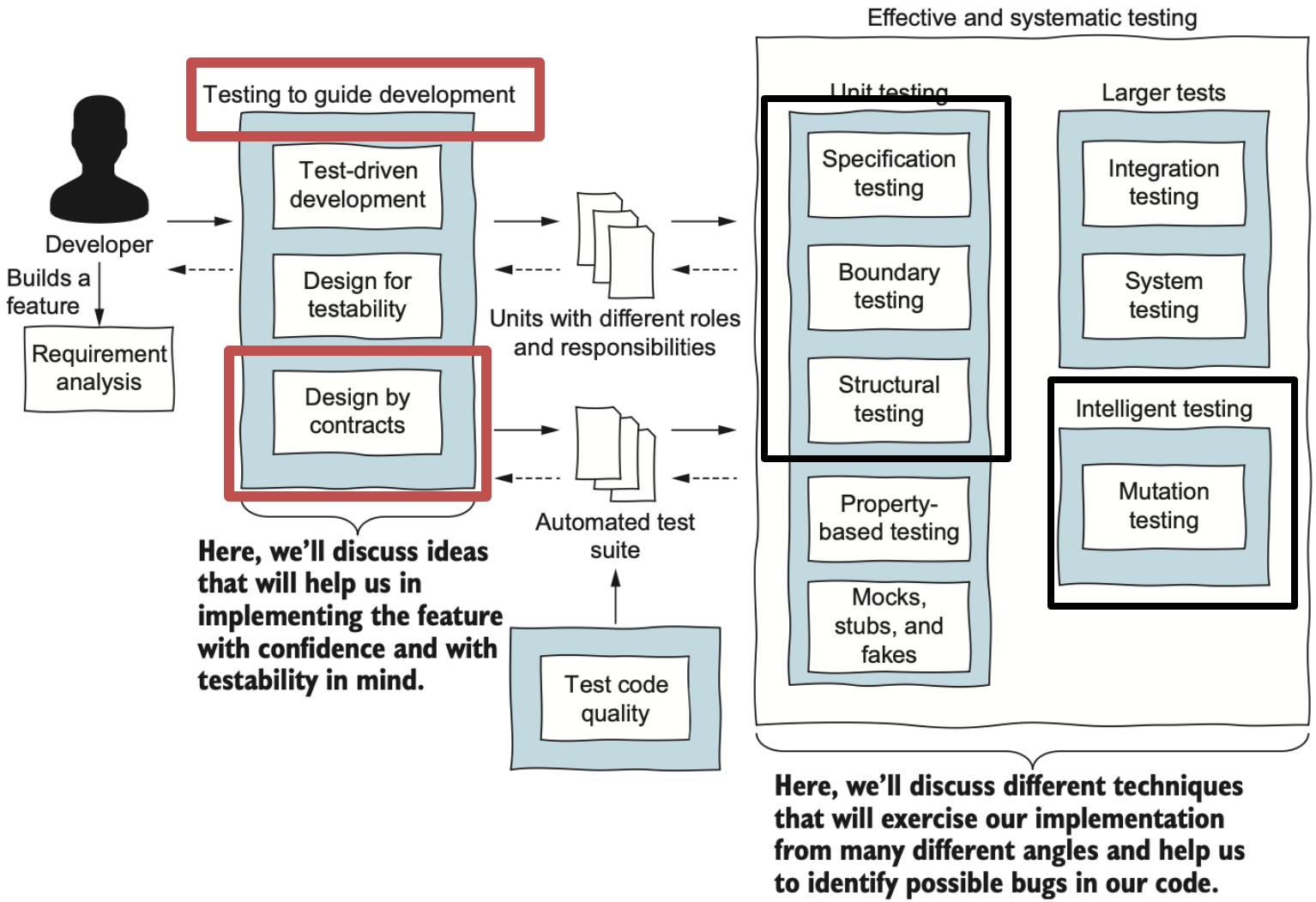
Dipartimento di Informatica - Università degli Studi di Bari
Via Orabona, 4 - 70125 - Bari
Tel: +39.080.5443270 | Fax: +39.080.5442536
serlab.di.uniba.it



What is designing contracts?

Designing contracts:
- design pre-conditions, post-conditions and invariants
- designing contracts is NOT validation

Effective and systematic testing (workflow)



Ex.: Tax calculator

TaxCalculator class handles calculating a specific tax. The calculation only makes sense for positive numbers.

Where to check this condition?

- In other classes calling TaxCalculator? (*add complexity to the caller classes*)
- Inside the class itself? (*defensive manner, system more resilient*)

The answer: define clear contracts for each class we develop. These contracts establish what the class requires as pre-conditions, what the class provides as post-conditions, and what invariants always hold for the class.

Design contracts

What classes and methods can and cannot handle and what they should do in case a violation happens.

Pre-conditions: what the method needs to function properly (ex. a positive number), should hold *before* method execution.

Post-conditions: what the method guarantees as outcomes (ex: does not return negative numbers), should hold *after* method execution.

Invariants: a condition that holds throughout the entire lifetime of an object or a data structure, must always hold *before* and *after* a method's execution.

Ex.: Tax calculator

```
public class TaxCalculator {
  public double calculateTax(double value) {

    if(value < 0) {
      throw new RuntimeException("Value cannot be negative.");
    }

    double taxValue = 0;

    // some complex business rule here...
    // final value goes to 'taxValue'

    if(taxValue < 0) {
      throw new RuntimeException("Calculated tax value
      ➥ cannot be negative.");
    }

     return taxValue;
  }
}
```

The pre-condition: a simple if ensuring that no invalid values pass

The post-condition is also implemented as a simple if. If something goes wrong, we throw an exception, alerting the consumer that the post-condition does not hold.

Ex.: Tax calculator

Making your pre- and post-conditions clear in the documentation is also fundamental and very much recommended.

```
/**
 * Calculates the tax according to (some
 * explanation here...)
 *
 * @param value the base value for tax calculation. Value has
 *              to be a positive number.
 * @return the calculated tax. The tax is always a positive number.
 */
public double calculateTax(double value) { ... }
```

The assert keyword

An `assert` is a necessary condition that must occur at some point in time. If this condition is not verified, the program is terminated.

Note that if you use Java's assert method to express pre/post-conditions, you must have assertions enabled in your JVM when you run the system.

The assert construct has two possible forms:

```
assert booleanExpression
assert booleanExpression : message
```

Ex.: `assert value >= 0 : "Value cannot be negative;`

This expression is equivalent to:

```
if(value<0)
throw new AssertException();
```

Ex.: Tax calculator

Assertions, differently from if statements, throw a generic exception.
Assert instruction can be disabled in production.

```
public class TaxCalculator {
  public double calculateTax(double value) {

    assert value >= 0 : "Value cannot be negative";

    double taxValue = 0;

    // some complex business rule here...
    // final value goes to 'taxValue'

    assert taxValue >= 0 : "Calculated tax value
    ➥ cannot be negative.";

    return taxValue;
  }
}
```

The same pre-condition, now as an assert statement

The same post-condition, now as an assert statement

Invariants

Invariants: a condition that holds throughout the entire lifetime of an object or a data structure, must always hold *before* and *after* a method's execution.

EX: Basket class

Basket class stores the products the user is buying from an online shop.

The class offers methods:

- `add(Product p, int quantity)`, which adds a product p a quantity number of times
- `remove(Product p)`, which removes the product completely from the cart

EX: Basket class

What are the pre and post-conditions?

```
public class Basket {
  private BigDecimal totalValue = BigDecimal.ZERO;
  private Map<Product, Integer> basket = new HashMap<>();

  public void add(Product product, int qtyToAdd) {
    // add the product
    // update the total value
  }

  public void remove(Product product) {
    // remove the product from the basket
    // update the total value
  }
}
```

We use BigDecimal instead of double to avoid rounding issues in Java.

Adds the product to the cart and updates the total value of the cart

Removes a product from the cart and updates its total value

EX: Basket class

- **Pre-conditions** for `add(Product p, int quantity)`:
  - product is not null (you cannot add a null product to the cart)
  - quantity is greater than 0 (you cannot buy a product 0 or fewer times)
- **Post-conditions** for `add(Product p, int quantity)`:
  - product is now in the basket.

EX: Basket class

- **Pre-conditions** for `add(Product p, int quantity)`:
  - product is not null (you cannot add a null product to the cart)
  - quantity is greater than 0 (you cannot buy a product 0 or fewer times)
- **Post-conditions** for `add(Product p, int quantity)`:
  - product is now in the basket.

```
public void add(Product product, int qtyToAdd) {
  assert product != null : "Product is required";
  assert qtyToAdd > 0 : "Quantity has to be greater than zero";

  // ...
  // add the product in the basket
  // update the total value
  // ...

  assert basket.containsKey(product) :
   "Product was not inserted in the basket";
}
```

Pre-condition ensuring that product is not null

Pre-condition ensuring that qtyToAdd is greater than 0

Post-condition ensuring that the product was added to the cart

EX: Basket class

- **Post-conditions** for `add(Product p, int quantity)`:
  - the new total value should be greater than the previous total value .

```
public void add(Product product, int qtyToAdd) {
  assert product != null : "Product is required";
  assert qtyToAdd > 0 : "Quantity has to be greater than zero";
  BigDecimal oldTotalValue = totalValue;

  // add the product in the basket
  // update the total value

  assert basket.containsKey(product) :
    "Product was not inserted in the basket";

  assert totalValue.compareTo(oldTotalValue) == 1 :
    "Total value should be greater than
    ➥ previous total value";
}
```

For the post-condition to happen, we need to save the old total value.

The post-condition ensures that the total value is greater than before.

Big- Decimals are recommended whenever you want to avoid rounding issues that may happen when you use doubles.

EX: Basket class

Basket class `remove(Product p)`:

- *Pre-conditions*: the product should not be null and the product to be removed needs to be in the basket
- *Post-condition*: after the removal, the product should no longer be in the basket

EX: Basket class

Basket class `remove(Product p)`:

- *Pre-conditions*: the product should not be null and the product to be removed needs to be in the basket
- *Post-condition*: after the removal, the product should no longer be in the basket

Pre-conditions: the product cannot be null, and it must exist in the basket.

```
public void remove(Product product) {
  assert product != null : "product can't be null";
  assert basket.containsKey(product) : "Product must already be in the
  ➥ basket";

  // ...
  // remove the product from the basket
  // update the total value
  // ...

  assert !basket.containsKey(product) : "Product is still in the
  ➥ basket";
}
```

Post-condition: the product is no longer in the basket.

Invariants

Invariants: a condition that holds throughout the entire lifetime of an object or a data structure, must always hold *before* and *after* a method's execution.

Ex. for Basket class: regardless of products being added to and removed from the basket, the total value of the basket should never be negative

```
public class Basket {
  private BigDecimal totalValue = BigDecimal.ZERO;
  private Map<Product, Integer> basket = new HashMap<>();

  public void add(Product product, int qtyToAdd) {
    assert product != null : "Product is required";
    assert qtyToAdd > 0 : "Quantity has to be greater than zero";
    BigDecimal oldTotalValue = totalValue;

    // add the product in the basket
    // update the total value

    assert basket.containsKey(product) : "Product was not inserted in
    ➥ the basket";
    assert totalValue.compareTo(oldTotalValue) == 1 : "Total value should
    ➥ be greater than previous total value";

    assert totalValue.compareTo(BigDecimal.ZERO) >= 0 :
      "Total value can't be negative."
  }

  public void remove(Product product) {
    assert product != null : "product can't be null";
    assert basket.containsKey(product) : "Product must already be in the
     basket";
    ➥
    // remove the product from the basket
    // update the total value

    assert !basket.containsKey(product) : "Product is still in the basket";

    assert totalValue.compareTo(BigDecimal.ZERO) >= 0 :
      "Total value can't be negative."
  }
}
```

The invariant ensures that the total value is greater than or equal to 0.

Can we do better than this?

The same invariant check for the remove

Invariants

The invariant checking may happen at the end of all the methods of a class!

To reduce duplication, create a method for such checks, such as the `invariant()` method to check at the end of every public method: after each method does its business (and changes the object's state), we want to ensure that the invariants hold.

Invariants

The invariant checking may happen at the end of all the methods of a class!

```
public class Basket {

  public void add(Product product, int qtyToAdd) {
    // ... method here ...
    assert invariant() : "Invariant does not hold";
  }

  public void remove(Product product) {
    // ... method here ...
    assert invariant() : "Invariant does not hold";
  }

  private boolean invariant() {
    return totalValue.compareTo(BigDecimal.ZERO) >= 0;
  }
}
```

Changing contracts

What happens if we change the contracts of our class?

Pre-condition for calculate-Tax method:
"value should be greater than or equal to 0"

vs

Pre-condition for calculate-Tax method:
"value should be greater than or equal to 100"

Changing contracts

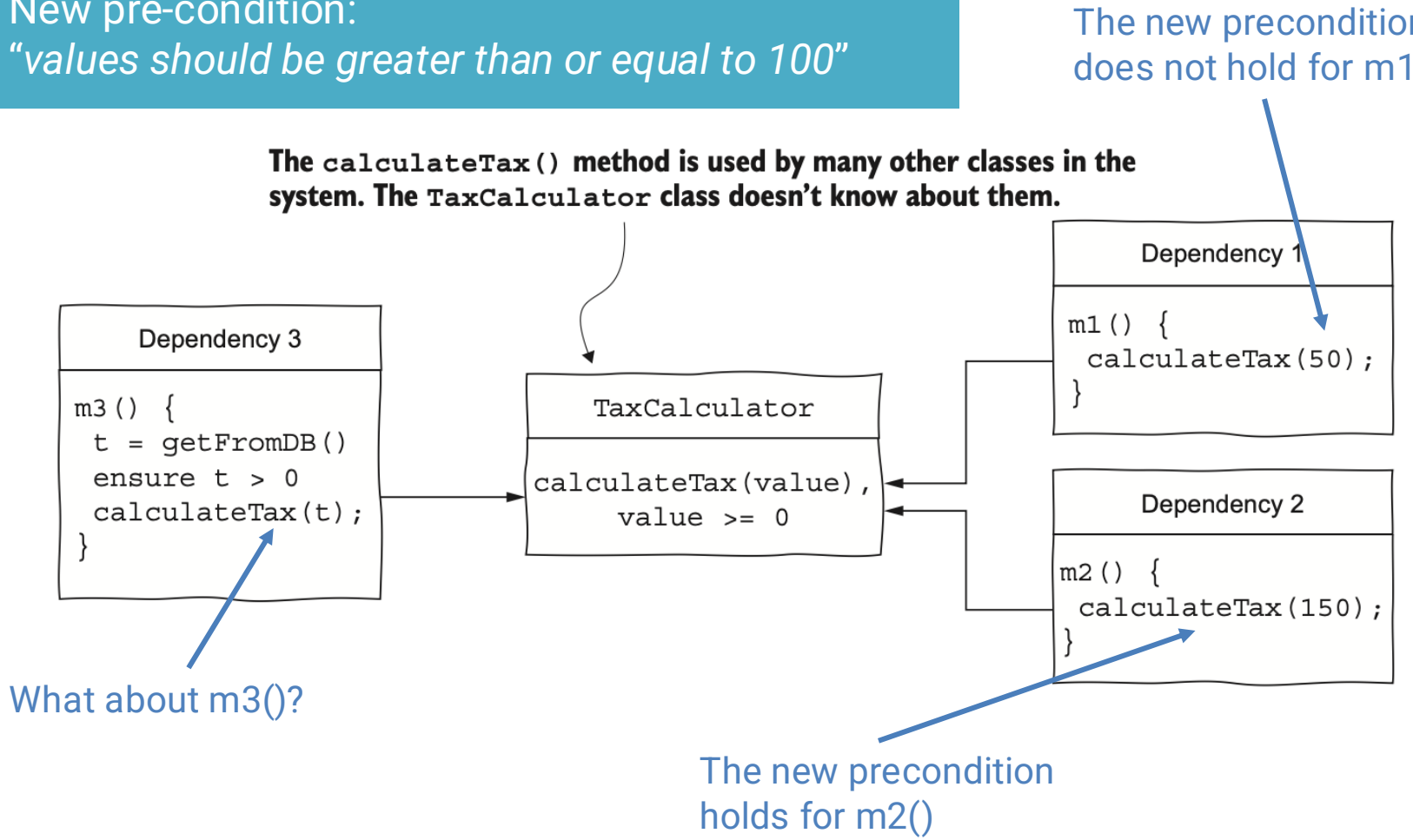
What happens if we change the contracts of our class?

It's important to understand the impact of a change!

You should not look at the change itself or at the class in which the change is happening, but at all the other classes (or dependencies) that may use the changing class.

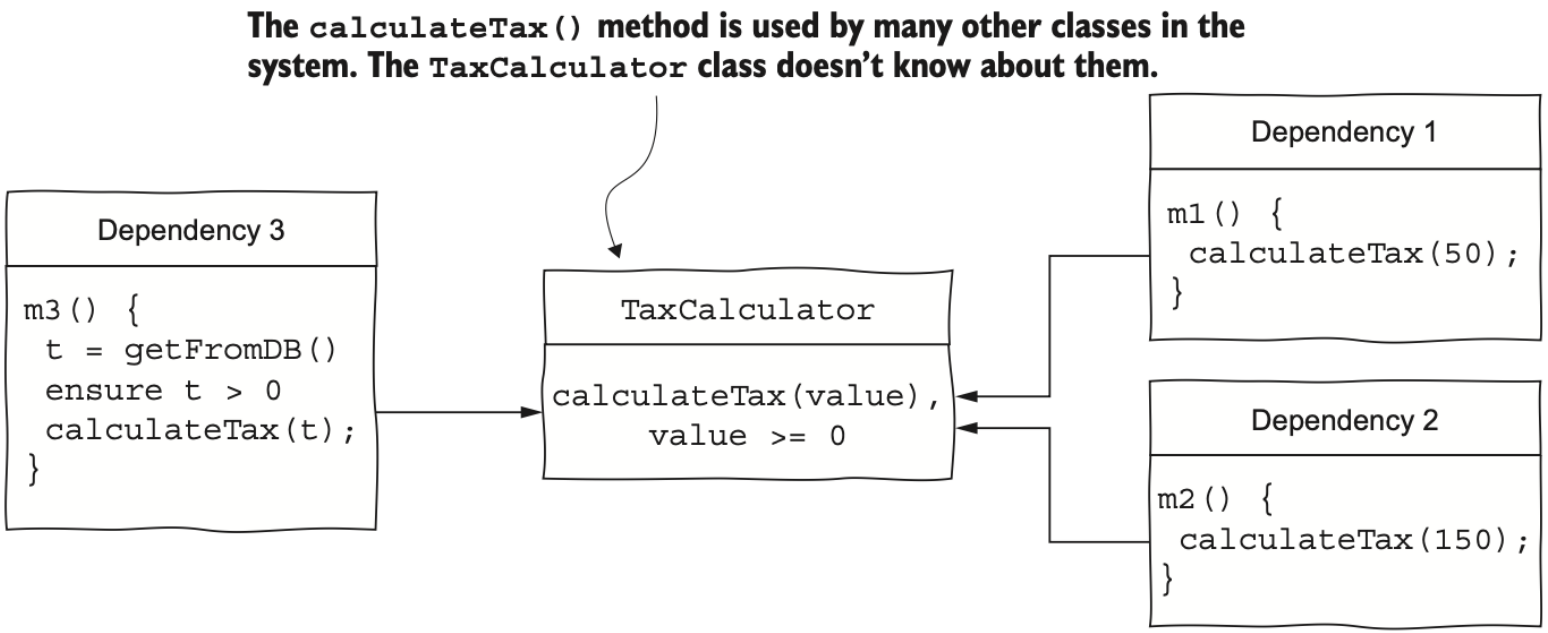
Changing contracts: analyze dependencies

New pre-condition:
"values should be greater than or equal to 100"



Changing contracts: analyze dependencies

New pre-condition:
"calculateTax() accepts also negative values"



Changing contracts: pre-conditions

Stronger pre-condition: it's more restrictive, our clients may break

Weaker pre-condition: it's less restrictive, we do not break the contract

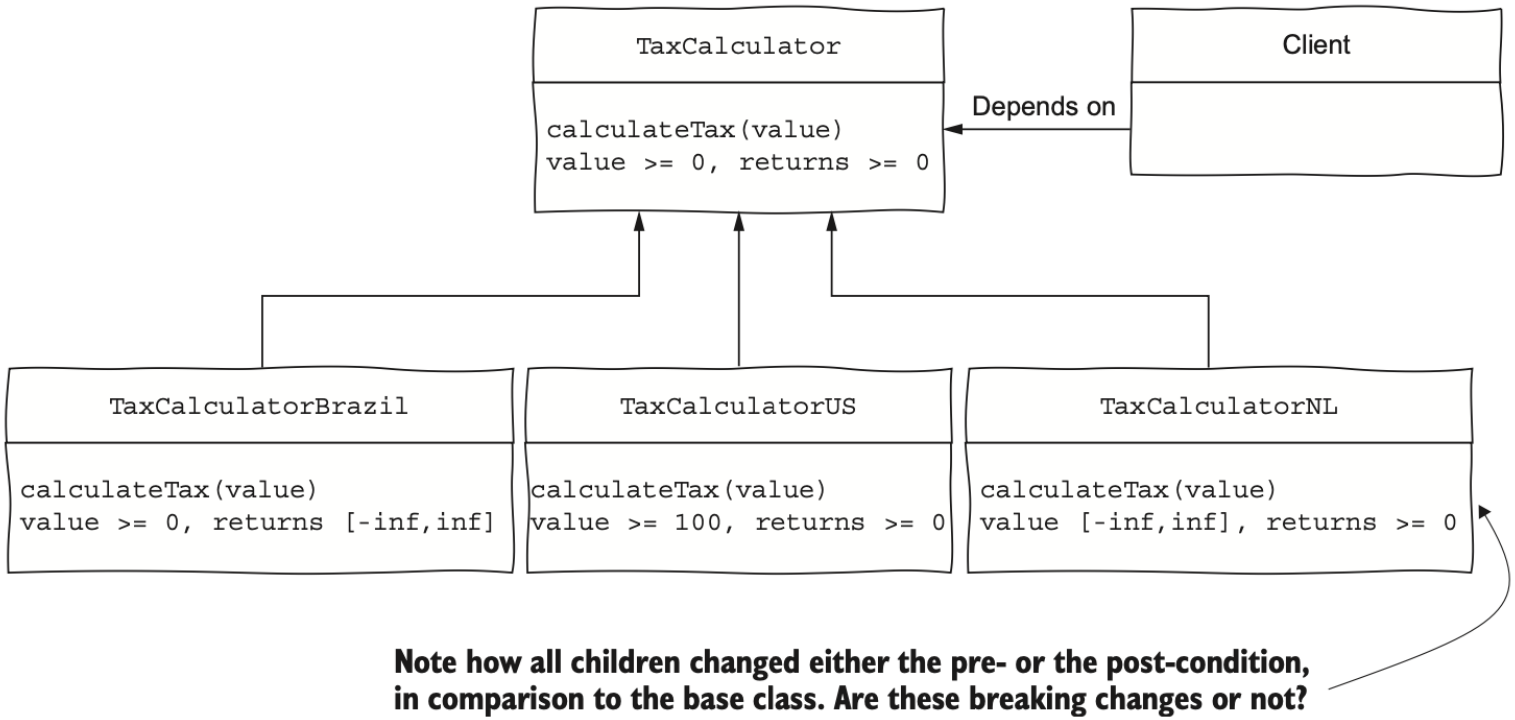
Changing contracts: post-conditions

Stronger post-condition: it's more restrictive, so it will prevent breaking changes in the dependencies

Weaker post-condition: it's less restrictive, our client may break

Example: `calculateTax` returns negative numbers

Inheritance and contracts



Inheritance and contracts

- `TaxCalculatorBrazil` class has a post-condition that the returned value is any number (weaker post-condition)
- `TaxCalculatorUS` class has pre-condition: “value greater than or equal to 100.” (stronger pre-condition)
- `TaxCalculatorNL` class has pre-condition: “it accepts any value.” (weaker pre-condition)

Inheritance and contracts

Whenever a subclass S (ex. `TaxCalculatorBrazil`) inherits from a base class B (ex. `TaxCalculator`):

- The *pre-conditions* of subclass S should be the same as or weaker (accept more values) than the pre-conditions of base class B.
- The *post-conditions* of subclass S should be the same as or stronger (return fewer values) than the post-conditions of base class B.

Reference: *Liskov substitution principle* (LSP)

Design-by-contract and testing

1) Contracts ensure bugs are detected early thanks to assertions

2) *Pre-conditions*, *post-conditions*, and *invariants* help developers about *what* to test. (Ex. `qty>0` pre-condition is something to exercise via unit, integration, or system tests).

3) Contracts do not replace (unit) testing: they complement it.

4) Design-by-contract is not a testing technique, is a design technique

Design-by-contract and testing

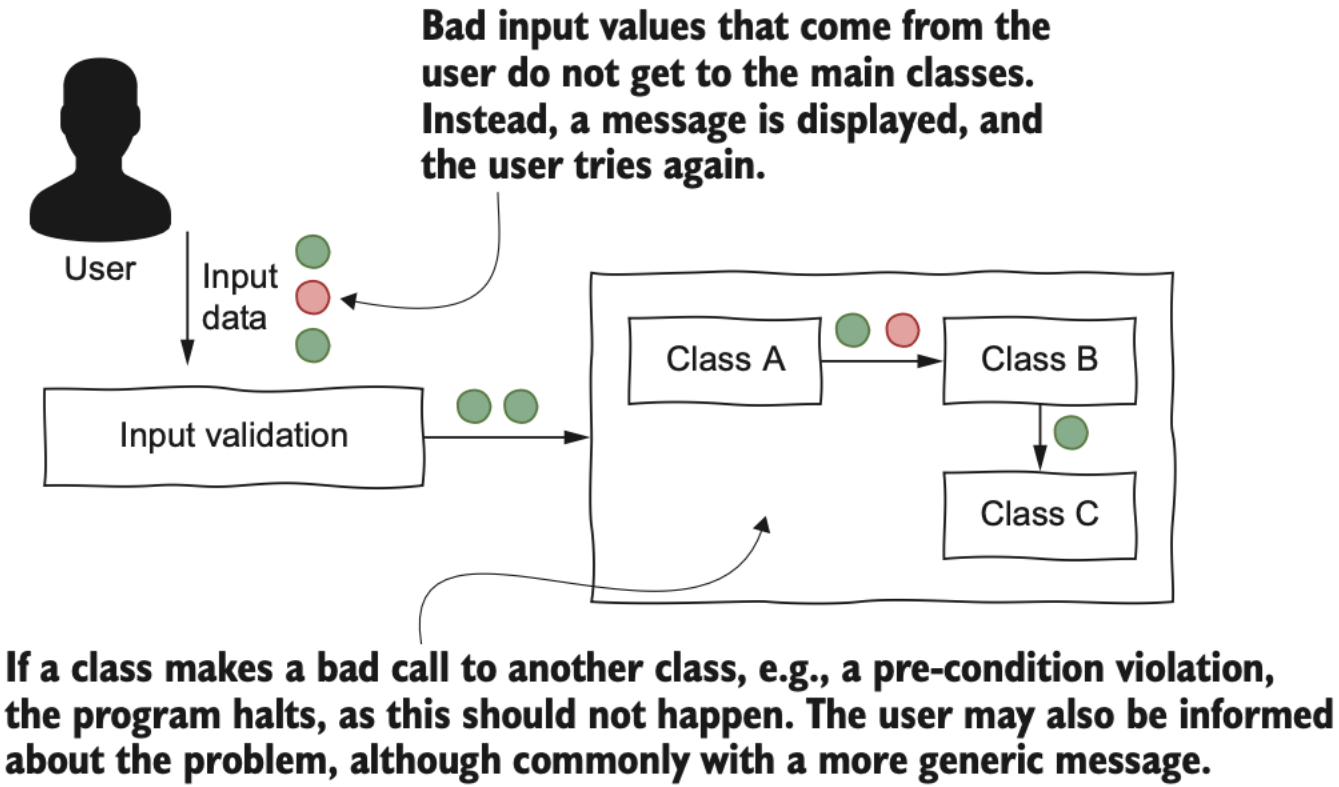
In a client-server application:

- The client should invoke the server properly (respecting its pre-conditions)
- If pre-conditions hold, the server can ensure to deliver the expected output (ensuring its post-conditions)

Weaker or strong pre-conditions?

- Weak pre-condition: easy for the client, an extra burden on the server, ex. the method must handle any invalid inputs (ex: null values)
- Strong pre-condition: The extra burden is now on the side of the client. The client must make sure it does not violate the pre-conditions of the method. This may require extra code.
- IT DEPENDS…

Contracts vs Validation



Exceptions or soft return value?

- Exception: *If it is behavior that should not happen*, and clients would not know what to do with it. That would be the case with the `calculateTax` method. (Ex.: If a negative value comes in we should halt the program rather than let it make bad calculations).
- Soft return value: *if it allow the client to keep working* (Ex.: a utility method that trims a string. A pre-condition of this method could be that it does not accept null strings. But returning an empty string in case of a null is a soft return that clients can deal with.)
- IT DEPENDS…

Design-by-contract: plus

It is not expensive and does not take a lot of time

It ensures proper communication among objects

Design-by-contract does not replace the need for testing.

Design-by-contract: recap

- Contracts ensure that **classes** can safely **communicate** with each other without surprises.
- Designing contracts means to explicitly defining the **pre-conditions**, **post-conditions**, and **invariants** of our classes and methods.
- Deciding to go for a ***weaker*** or a ***stronger*** **contract** is a contextual decision. Both have advantages and disadvantages.
- ***Validation*** and ***contract checking*** are different things with different objectives. Both should be done.
- Whenever **changing a contract**, we need to reflect on the **impact** of the change.

Design-by-contract: Exercise

A method `M` belongs to a class `C` and has a pre-condition `P` and a post-condition `Q`. Suppose that a developer creates a class `C'` that extends `C` and creates a method `M'` that overrides `M`.

Which one of the following statements correctly explains the relative strength of the pre- (`P'`) and post-conditions (`Q'`) of the overridden method `M'`?

A `P'` should be equal to or weaker than `P`, and `Q'` should be equal to or stronger than `Q`.

B `P'` should be equal to or stronger than `P`, and `Q'` should be equal to or stronger than `Q`.

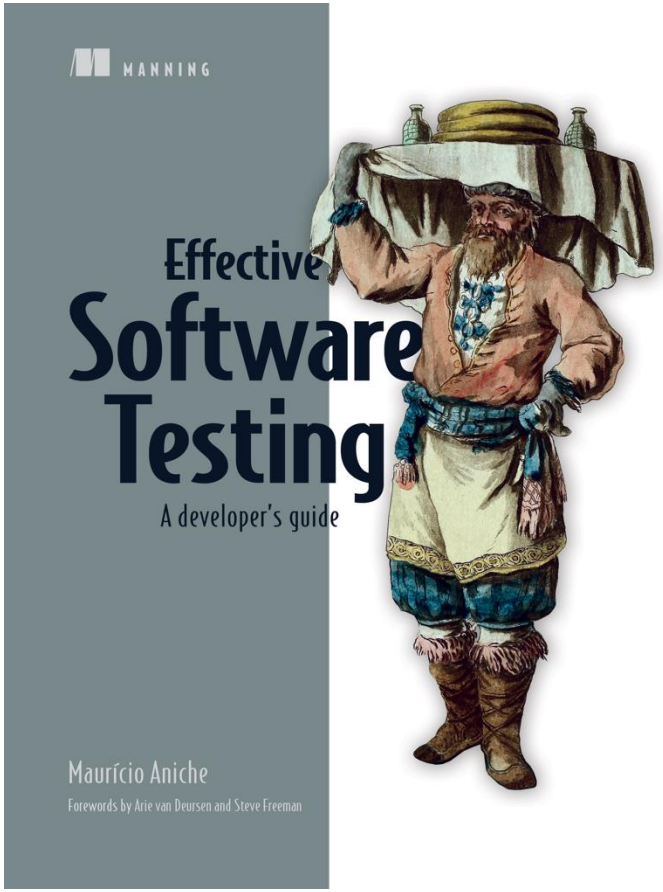
C `P'` should be equal to or weaker than `P`, and `Q'` should be equal to or weaker than `Q`.

D `P'` should be equal to or stronger than `P`, and `Q'` should be equal to or weaker than `Q`.

Reference book:

Effective Software Testing. A developer's guide. Mauricio Aniche. Ed. Manning. (**Chapter 4**)

.



Azzurra Ragone

Department of Computer Science- Floor VI – Room 616
Email: azzurra.ragone@uniba.it