

Design by Contract

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Monday, Jan. 23, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

Last Class

- 1 Intentional (Bad) Design
- 2 Reacting, Designing and Adapting for Change
- 3 Encapsulation
- 4 Information Hiding
- 5 Separation of Concerns
- 6 Reuse
- 7 Design Questions to Ask

Homework Assignment

- 1 Watch the various videos posted on your lab Canvas page.
- 2 Project 1 is posted, so please start as soon as possible!

Design by Contract (DBC)

- ▶ Articulated clearly in the 1980s (Bertrand Meyer)
- ▶ Design-by-contract has become the standard policy governing **separation of concerns** across modern software engineering
- ▶ This is how software components are really used. . .

Contracts

- ▶ We use contracts in the real world to define responsibilities
- ▶ **For Example:** Leasing Contracts
 - ▶ The tenant is responsible for paying rent and following the rules of the lease
 - ▶ The landlord is responsible for providing and maintaining the house
 - ▶ They sign a contract to clarify these responsibilities

Idea of a Contract

- ▶ 2 Roles¹
 - 1 Implementer
 - 2 Client
- ▶ Implementers and clients communicate through contracts
- ▶ Contracts “hide information” by design
 - ▶ Implementation details aren't necessary, just follow the contracts
- ▶ Contracts may be **Formal** or **Informal**
 - ▶ BUT they must be understandable to clients
 - ▶ SO THAT the hidden information isn't needed to understand how to use the code

¹When you write code, you might go between roles

Contracts in Software Engineering

- ▶ Special comments in the code
 - ▶ It is just a communication tool via comments
 - ▶ Which compilers ignore comments, so they are not enforced
 - ▶ **AKA Documentation!**
- ▶ Defines the responsibilities of each role, essentially:
 - ▶ what code is the *implementer* responsible for writing
 - ▶ what code is the *client* (AKA *user*) responsible for writing

Precondition Contract

- ▶ A **precondition** (aka a *requires clause*) defines the **client's** responsibility
 - ▶ Defines what the client must do before calling the code
 - ▶ Steps that need to have already happened
 - ▶ Can be related to the parameters
 - ▶ Current state of the object
- ▶ **Need to ask the question:**
What would cause this code to fail?
- ▶ Not all operations will have a precondition, but most will

Why Use Preconditions?

- ▶ Efficiency!
- ▶ In the absence of preconditions, implementations will have to do error checking.
- ▶ Those checks will happen every time an operation is called, whether or not checks are needed!
- ▶ Those checks may happen in situations where the operation can not deal with the error

Simple Example

- ▶ Example function:

```
private static int func (int x) {...}
```

- ▶ Suppose for `func` to work properly `x` must be between 1 and 4.
- ▶ If the code for `func` did the checking, it would perform a check even on the following call:

```
int ans = func(3);
```
- ▶ For the above call, neither the caller nor the code needs to do any checking!

Simple Example

- ▶ Example function:

```
private static int func (int x) {...}
```

- ▶ Suitable precondition for `func`
 - ▶ $1 \leq x \leq 4$

Another Example

- ▶ Operation (or method) to binary search if an item is in an array
 - ▶ **Question:** What is the precondition?

Another Example

- ▶ Operation (or method) to binary search if an item is in an array
 - ▶ **Question:** What is the precondition?
 - ▶ **ANSWER:** The array must be sorted
- ▶ **Question:** What if the function checked the precondition on it's own?

Another Example

- ▶ Operation (or method) to binary search if an item is in an array
 - ▶ **Question:** What is the precondition?
 - ▶ **ANSWER:** The array must be sorted
- ▶ **Question:** What if the function checked the precondition on it's own?
 - ▶ **ANSWER:** It would take *linear* time to check that the array is sorted, though it'd take only *log* time to search!
 - ▶ Defeating the entire purpose of an efficient search

Yet Another Example

- ▶ Integer division operation
 - ▶ Does it have an (implicit) precondition?
 - ▶ Why?

Yet Another Example

- ▶ Integer division operation
 - ▶ Does it have an (implicit) precondition?
 - ▶ Why?
- ▶ **ANSWER:** Yes, divisor should not be 0!
- ▶ Why not have the division code check?

Another reason for the precondition

Input Validation

- ▶ We have a function `foo` that takes in `int x`, and has to be greater than 0
- ▶ With a precondition:
 - ▶ Client is responsible for `x`
 - ▶ Client checks before calling `foo`!

Another reason for the precondition

- ▶ Input Validation
 - ▶ We have a function `foo` that takes in `int x`
 - ▶ `x` has to be greater than 0
- ▶ With a precondition:
 - ▶ Client is responsible for `x`
 - ▶ Client checks (if needed) before calling `foo`!

Another reason for the precondition

- ▶ Input Validation
- ▶ Without a precondition:
 - ▶ Client does not check
 - ▶ Implementer does
- ▶ What does `foo` do if `x <= 0`?
 - 1 Ask the user for new input?
 - ▶ Separation of concerns
 - 2 Return `false` to show the function didn't work?
 - ▶ What if the function needs to return a non-Boolean value?
 - 3 Return a special value to indicate an error?
 - ▶ Now client has to check
 - ▶ What special (aka magic) value will you use?
 - 4 Throw an exception
 - ▶ Which the client will need to trap for (`catch`)

Postcondition Contract

- ▶ A **postcondition** (aka an *ensures clause*) that characterizes the implementer's responsibility (aka, your code)
 - ▶ What the code does
- ▶ **Need to ask the question:**
What will be true **AFTER** the code is finished?

Why Use Postconditions?

- ▶ Tells the client what the code does
 - ▶ Return values
 - ▶ Events
 - ▶ State of the object
- ▶ Allows us to know information about our variables
 - ▶ May need it for later preconditions
 - ▶ **Example:** postcondition of sorting can help with precondition of binary search
- ▶ All of this should be done **without needing** any details about the implementation

Separating Concerns

- ▶ Gives responsibilities to each role
- ▶ Tells you what you can assume
- ▶ **Implementer**
 - ▶ Assumes precondition is true
 - ▶ Responsible for postcondition
- ▶ **Client**
 - ▶ Responsible for precondition
 - ▶ Assumes postconditions will be true
- ▶ What happens when either one doesn't live up to their responsibilities?

Last little bit

- ▶ End users are not clients
 - ▶ May not understand responsibilities such as preconditions
 - ▶ Verification of user input must be done
- ▶ Contracts are comments and not actual code
- ▶ Contracts pass input validation responsibility up the layers of code
 - ▶ Forces the implementers of the UI to validate input

Role of Contracts

What does this method call do? How do you know?

```
double a, b;  
...  
double c = sqrt(a*a + b*b, 0.001);
```

Example of a Contract

```
/**
 * <description omitted for brevity...>
 *
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND [x and epsilon values don't change] AND
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

Note: A Java comment that starts with the symbols: `/**` is called a **Javadoc comment**. This goes before the method header.

- ▶ The standard documentation technique for Java is called **Javadoc**
- ▶ You place special **Javadoc comments** enclosed in `/** ... */` in your code, and the **Javadoc tool** generates nicely formatted web-based documentation from them

- ▶ The resulting documentation is known as the **API** (**application programming interface**) for the Java code to which the Javadoc tags are attached
- ▶ The word **interface** has two related, but distinct meanings:
 - ▶ A unit of Java code that contains Javadoc comments used to produce documentation
 - ▶ The resulting documentation

Example of a Contract

```
/**
 * <description omitted for brevity...>
 *
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND [x and epsilon values don't change] AND
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

Note: The **Javadoc tag** *@param* is needed for each formal parameter; you describe the parameter's role in the method.

Example of a Contract

```
/**
 * <description omitted for brevity...>
 *
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND [x and epsilon values don't change] AND
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

Note: The **Javadoc** tag *@return* is needed if the method returns a value; you describe the returned value.

Example of a Contract

```
/**
 * <description omitted for brevity...>
 *
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND [x and epsilon values don't change] AND
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

Note: The Javadoc tag *@pre* introduces the precondition for the `sqrt` method.

Example of a Contract

```
/**
 * <description omitted for brevity...>
 *
 * @param x number to take the square root of
 * @param epsilon allowed relative error
 * @return the approximate square root of x
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND [x and epsilon values don't change] AND
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

Note: The **Javadoc** tag *@post* introduces the postcondition for the `sqrt` method.

Abbreviated Javadoc

For this course:

- ▶ Code you see *in these slides* will *not* have the Javadoc tags *@param*, *@return* and doesn't provide a description of the method
- ▶ “keywords” in the Javadoc and mathematics will be bold-faced for easy reading
 - ▶ This allows you to focus on the contract content: the preconditions and postconditions themselves
- ▶ **However**, in lab assignments, homework assignments and exams, you **will need** to provide the complete documentation with the contracts.

Example of a Contract (Abbreviated)

```
/**  
 * ...  
 * @pre  
 *  $x \geq 0$  AND  $\epsilon > 0$   
 * @post  
 *  $\text{sqrt} \geq 0$  AND [x and epsilon values don't change] AND  
 * [sqrt is within relative error epsilon  
 * of the actual square root of x]  
 */  
private static double sqrt(double x,  
    double epsilon)
```

Example of a Contract (Abbreviated)

Preconditions

```
/**
 * ...
 * @pre
 *  $x \geq 0$  AND  $\epsilon > 0$ 
 * @post
 *  $\text{sqrt} \geq 0$  AND [x and epsilon values don't change] AND
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

Note: The preconditions indicates that the *arguments* passed in for the *formal parameters*: `x` and `epsilon` both must be positive before a client may call `sqrt`.

Example of a Contract (Abbreviated)

Preconditions

```
/**  
 * ...  
 * @pre  
 *  $x \geq 0$  AND  $\epsilon > 0$   
 * @post  
 *  $\text{sqrt} \geq 0$  AND [x and epsilon values don't change] AND  
 * [sqrt is within relative error epsilon  
 * of the actual square root of x]  
 */  
private static double sqrt(double x,  
    double epsilon)
```

Note: The precondition is a *statement about the models of the arguments*; Here, it is a *formal* mathematical statement about mathematical **reals**.

Example of a Contract (Abbreviated)

Preconditions

```
/**
 * ...
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND [x and epsilon values don't change] AND
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

Note: This is the postcondition, indicating that the *return value* from `sqrt` is non-negative and ... what does the rest say?

Example of a Contract (Abbreviated)

Preconditions

```
/**
 * ...
 * @pre
 *  $x \geq 0$  AND  $\epsilon > 0$ 
 * @post
 *  $\text{sqrt} \geq 0$  AND [x and epsilon values don't change] AND
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

Note: The first part of the postcondition here is written in *mathematical* notation; it is not program code!
The second part – inside [...] – is written in English.

Contracts Use

- ▶ Parameter names
- ▶ Method name
 - ▶ Returned value
- ▶ Numbers/mathematical notation
- ▶ Class variable names
- ▶ `#var`
 - ▶ The original value stored in `var`

Contracts

- ▶ Formal is better (more concise)
 - ▶ Written only using mathematics
 - ▶ A mathematical Boolean expression
- ▶ Am I meeting the precondition?
 - ▶ Does it evaluate to true?
- ▶ What will happen in the code?
 - ▶ Postcondition
- ▶ Am I meeting the postcondition?
 - ▶ Will the postcondition evaluate to true?
 - ▶ **NOTE:** For all inputs that meet the precondition

Reasoning: Tracing Tables

Code	State
	$y = 76.9$
<code>y = sqrt(4.0, 0.01);</code>	
	$y = 2.0$

Reasoning: Tracing Tables

Code	State
	$y = 76.9$ $z = 4.0$
<code>y = sqrt(z, 0.01);</code>	
	$y = 2.0$ $z = 4.0$

Reasoning: Tracing Tables

Code	State
	$y = 76.9$ $z = 4.0$
<code>y = sqrt(z, 0.01);</code>	
	$y = 2.0$ $z = 4.0$

Question 1: From the contract of `sqrt`, do we know that $y = 2.0$ instead of $y = -2.0$?

Reasoning: Tracing Tables

Code	State
	$y = 76.9$ $z = 4.0$
<code>y = sqrt(z, 0.01);</code>	
	$y = 2.0$ $z = 4.0$

Question 2: From the contract of `sqrt`, do we know that $y = 2.0$ instead of $y = 1.9996$?

A Partly Informal Contract

```
/**
 * ...
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND [x and epsilon values don't change] AND
 * [sqrt is within relative error epsilon
 * of the actual square root of x]
 */
private static double sqrt(double x,
    double epsilon)
```

A Formal Contract

```
/**
 * ...
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND x = #x AND epsilon = #epsilon AND
 * |sqrt - x^(1/2)| / x^(1/2) <= epsilon
 */
private static double sqrt(double x,
    double epsilon)
```

A Formal Contract

```
/**
 * ...
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND x = #x AND epsilon = #epsilon AND
 * |sqrt - x^(1/2)| / x^(1/2) <= epsilon
 */
private static double sqrt(double x,
    double epsilon)
```

Note: We can, in this formal setting, easily substitute 4.0 for `x`, 0.01 for `epsilon`, and either 2.0 or 1.9996 for `sqrt`... and is the postcondition *true* in either case?

A Formal Contract

```
/**
 * ...
 * @pre
 * x >= 0 AND epsilon > 0
 * @post
 * sqrt >= 0 AND x = #x AND epsilon = #epsilon AND
 * |sqrt - x^(1/2)| / x^(1/2) <= epsilon
 */
private static double sqrt(double x,
    double epsilon)
```

Note: We can, in this formal setting, easily substitute 4.0 for `x`, 0.01 for `epsilon`, and either 2.0 or 1.9996 for `sqrt`... and is the postcondition *true* in either case?

Answer: Yes!

Using Java `assert` statements

- ▶ Just during development

- ▶ Identify bugs

- ▶ **Example:**

```
assert x > 0.0 : "Violation of precondition x > 0";
```

- ▶ Boolean condition
 - ▶ What to print to terminal if true
- ▶ This checking can be turned off (to avoid inefficiency in production software) using the `-ea` argument to the JVM

- ▶ **Wikipedia: Design by Contract**

https://en.wikipedia.org/wiki/Design_by_contract

Javadoc on School of Computing Unix Machines

Running Javadoc on School of Computing Unix Machines

- ▶ Javadoc is a documentation generator and we will be using it to generate API documentation along with our contracts in HTML format
 - ▶ It comes installed along with the JDK
 - ▶ Usage of HTML is allowed
 - ▶ Can add styling via CSS
- ▶ The instructions on the following slides have been tested on the School of Computing Unix machines
 - ▶ ...but may work on your own computers (with some minor tweaks)
- ▶ **See:** <https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html>

Running Javadoc on School of Computing Unix Machines

- ▶ Create a new directory called `doc` right next to your Java top-level package²
- ▶ By default, Javadoc won't understand our contract tags, so we will need to let it know about them.
- ▶ **Example:** Lab 1
 - ▶ `javadoc -tag pre -tag post -tag invariant -tag constraints -tag correspondences -d doc -subpackages cpsc2150`
- ▶ This will populate the `doc` folder with all necessary to render your API (Javadoc + contracts)

²This is not strictly required, but does allow us to simplify the command

Homework Assignment

- 1 Watch the various videos posted on your lab Canvas page.
- 2 Project 1 is posted, so please start as soon as possible!

Summary

- 1 Design by Contract (DBC)
- 2 Preconditions and Postconditions
- 3 Javadocs and APIs
- 4 Introduction to Tracing Tables
- 5 Running Javadoc on School of Computing Unix Machines