

# Design-by-Contract

Hey! This is Dr. Holly.

Thanks for joining me for another talk from our series “Notes on Engineering Software”.

Today’s focus is on the concept called Design by Contract.

And as usual we’re going to be introducing a number of new concepts and terms, so be ready to pause the video and take a few notes.

The main idea behind Design by Contract – is that there is a contract that governs the interaction between a client program and an operation called by the client program.

# Main Points

- *Design-by-Contract*
  - Mathematical (formal) and informal models
  - Contracts for operations
  - Assignment of responsibility in Design-by-Contract
  - How to reason about software using a contract

2

1. With Design by Contract we utilize both formal mathematical models and informal models to write down the contract that must govern how an operation is to be utilized.
2. With these contracts, we can then create an assignment of responsibilities to the client and the operation that provides a service
3. This contract and assignment of responsibilities allows us to precisely reason about the interactions of the client and operation – no guessing involved

# Overview of Design-by-Contract

*Design-by-contract* is also known as *contract programming*

- Separation of Concerns:
  - “what it does” (captured by the contract)
  - “how it does it” (captured by the code)
- Design-by-contract is *the standard policy* governing “separation of concerns” across the engineering of software

3

1. Contract programming is another name for Design by Contract
2. And in Contract Programming or Design by Contract, when we talk about Separation of Concerns for a piece of software, the two concerns are:
  - “what it does” – which is captured by the contract
  - “how it does it” – which is captured by the code
3. Design by Contract is a policy that governs Separation of Concerns in the engineering of software

# Mathematical Models

- Each *variable* in the program has a *type*  
Examples: `int`, `double`, ...
- Each type has a *mathematical model*: think of any variable of that type as having a value from its mathematical model's mathematical space/domain

4

1. Variables in our programs are declared from some type, for example an `int` in C++ or Java
2. In a programming language a type like `int` has an implementation, for example, the integer 5 stored in variable declared as an `int` will usually be implemented by storing a very specific pattern of bits in the computer's memory – this is the “how it does it” part of Separation of Concerns
3. But the “what it does” part is couched in the mathematical model. For example, for type `int`, we're going to use the mathematical Integers, the ones we have studied since grade school. We'll utilize our knowledge of math Integers to reason about how operations like addition and subtraction

# Examples: Mathematical Models

Programming type and mathematical model

- `int` and *integer*
- `double` and *real*
- etc.

5

1. Here are two examples of mathematical models for two different programming types.
2. As we talked about in the previous slide, we'll use math integers to reason about variables of type `int`

# Structure of an Operation Contract

- Each operation has:
  - A *precondition* (*requires clause*)
  - A *postcondition* (*ensures clause*)

6

1. Now let's look at how we capture a contract for an operation
2. The operation's precondition is written with what is known as a requires clause
3. The operation's postcondition is written using the ensures clause

# Operation Contract - Precondition

- It captures in a mathematical expression what must be true about the state of the program when the operation is called
- Characterizes the responsibility placed on the client program that *calls (uses)* that operation

7

1. The precondition mathematically captures what must be true at the time the operation is called.
2. And making sure these conditions are satisfied at the time of the call is the responsibility of the client program.



# Operation Contract - Postcondition

- It captures in a mathematical expression what will be true about the state of the program when the operation terminates
- Characterizes the responsibility of the code that *implements* that operation

8

1. The postcondition mathematically captures what will be true at the time the operation terminates.
2. And making sure these conditions are satisfied is the responsibility of the code that implements the operation.



# The Execution of a Contract in Two Situations

1. The precondition is *true* when an operation is called
2. The precondition is *not true* when the operation is called

9

1. When the client program calls an operation, the precondition could evaluate to two different values:
  - It could evaluate to true
  - Or it could evaluate as not true
2. The next two slides look at both these situations

## Situation #1 – Precondition is True

- Then the operation will *terminate* and the postcondition will be *true* when it returns control back to the caller
- Terminate means: return to the calling program

10

1. If the client program calls the operation when the precondition is true, then the operation will terminate, and the postcondition will be true when control returns back to the caller
2. In this situation, the client program has followed the contract.

## Situation #2 – Precondition is Not True

- Then the operation may do anything, including not terminate

11

1. If on the other hand, the client calls the operation when the program is in a state where the precondition is not true then the called operation may do anything including not terminate.
2. In this situation, the client program has violated the contract, which then frees the called program from adhering to the contract.

# Responsibilities & Rewards

## Client View

- Responsibility:

Making sure the *precondition* is true when an operation is called is the responsibility of the *client*

- Reward:

The *client* may assume the *postcondition* is true when the operation returns

12

1. Contract programming has its responsibilities, it also has its rewards
  - The client has the responsibility to make sure the precondition is true at the time of the call
  - The reward is that the client may assume the postcondition is true when the called operation returns

# Responsibilities & Rewards Implementer View

- Responsibility:

Making sure the *postcondition* is true when an operation returns is the responsibility of the *implementer* of that operation

- Reward:

The *implementer* may assume the *precondition* is true when the operation is called

13

1. Making sure the postcondition is true is the responsibility of the operation's implementer
2. The reward is that the implementer may assume the precondition is true when the operation is called

## Recall: An Operation

- Has zero or more *formal parameters* of various types — placeholders for the information passed to the call
- Returns a value of a particular *return type* to the calling program; or, returns nothing, denoted by a return type of `void`

14

1. Now let's look at an example, but first we have to recall the structure of an operation's header
2. An operation has zero or more formal parameters that allow information to pass between the client and the called operation
3. The operation may also return a value to the caller or it may return nothing
4. If it returns a value, then the operation has a return type
5. If it returns nothing, then the return type is void

# Example: Contract as a Comment

```
int plusOne(int z)
// requires: z < INT_MAX
// ensures: plusOne = #z + 1 and z = #z
```

Operation plusOne has:

- 1 formal parameter of type int
- The return type is “int”

15

1. In this example, the operation is called plusOne
  - It has one formal parameter named ‘z’ of type int
  - It has a return type of int
2. The requires clause states that the value passed by the caller to the formal parameter ‘z’ must be strictly less than INT\_MAX.
3. INT\_MAX is defined in programming languages like C++ to be the maximum size integer that can be stored in the int programming type.
4. The ensures clause for plusOne states that the value returned will be equal to 1 plus the value that was originally passed to plusOne.



# Specifics About Reading an Operation's Contract

An operation's *contract* appears in the requires and ensures clauses and refers to:

- its formal parameters by their names
- the name of the operation, if it returns a value, i.e., has a non-void return type

16

1. The operation's contract appears in the requires and ensures clauses
2. It refers to the formal parameters using their names to place conditions on the formal parameters and
3. It refers to the name of the operation to place conditions on the value being returned

# Specifics About Reading an Operation's Contract

Two different times must be referenced in the contract

- right before the call and
- at the time the called operation terminates
- a name with # prepended indicates the value in the formal parameter at the time of the call
- a name without # prepended indicates the value at the time the call terminates

17

1. The contract must be able to refer to values of variables at two different times:
  - Right before the call
  - At the time the called operation terminates
2. To refer to a value of a formal parameter at the time of the call (or the incoming value), a # is prepended to the formal parameter name
3. A name without the # prepended indicates the value at the time the call terminates (or the outgoing value)

# Using an Operation's Contract

To determine whether the precondition and postcondition are true for a particular client call...

Perform a substitution:

Substitute the values of the arguments for the respective formal parameters

Substitute the value of the result returned by the operation for the operation's name

18

1. So how do we use the contract to reason about the correctness of our program?
2. At the time of a call we must substitute into the requires clause the values of all the arguments being passed to the called operation
3. At termination of the call we must substitute the value being returned by the operation

# Reasoning: Tracing Table #1

<i>Code</i>	<i>State</i>
	x = 4 y = 79
y = plusOne(x);	
	x = ??? y = ???

19

1. Here is an example using a tracing table.
2. The tracing table has two columns, column one shows the code to be executed, and column two allows us to record the values of variables involved in the executed code – this is also known as program state.
3. In this example, you can see that the program state before the call indicates x = 4 and y = 79
4. We're going to use the contract for plusOne to make sure this client code is following the contract and also to predict what the value of x and y are after the call terminates.

# Reasoning: Tracing Table #1

<i>Code</i>	<i>State</i>
	x = 4 y = 79
y = plusOne(x);	
	x = 4 y = 5

20

1. In this slide the state after the call to plusOne has been filled in with x = 4, and y = 5. Assuming 4 is less than INT\_MAX, we know from plusOne's ensures clause that it will return 1 + 4 or 5, and that the value in 'x' won't be changed.

# Questions: Tracing Table #1

Code	State
Question #1: did the client satisfy the requires clause?	x = 4 y = 79
y = plusOne(x);	
Question #2: does the client know the following: y = 5?	x = 4 y = 5

```
int plusOne(int z)
// requires: z < INT_MAX
// ensures: plusOne = #z + 1 and z = #z
```

21

1. Now we're going to utilize the contract to determine the answer to the two questions shown here.
2. Question #1 – did the client satisfy the requires clause?
3. Question #2 – does the client know that y = 5 after the call terminates?

## Answers: Tracing Table #1

Code	State
Question #1: did the client satisfy the requires clause?	x = 4 y = 79
y = plusOne(x);	

First:

- substitute the value 4 for z in the requires clause
- 4 is stored in 'x' which is the actual parameter to plusOne

```
int plusOne(int z)
// requires: z < INT_MAX
```

22

1. To answer Question #1 we must substitute the value 4 for the formal parameter 'z' found in the requires clause



## Answers: Tracing Table #1

Code	State
Question #1: did the client satisfy the requires clause?	x = 4 y = 79
y = plusOne(x);	

Second:

- evaluate the requires clause after the substitution
- So the answer is “yes” to Question #1,  $4 < \text{INT\_MAX}$

```
int plusOne(int z)
// requires: 4 < INT_MAX
```



23

1. Here you'll see that we have substituted 4 into the requires clause giving us the Boolean expression  $4 < \text{INT\_MAX}$
2. Now we evaluate requires clause – here we'll assume  $\text{INT\_MAX}$  is for a 32 bit machine and is defined to be approximately 2.1 billion
3. So in this case the requires clause evaluates to true, meaning that the client satisfied the requires clause, and the answer to the first question is “yes”

# Questions: Tracing Table #1

Code	State
<code>y = plusOne(x);</code>	
Question #2: does the client know the following: <code>y = 5</code> ?	<code>x = 4</code> <code>y = 5</code>

First:

- substitute the value 4 for #z in the ensures clause
- #z represents the incoming value to plusOne, which is 4

```
int plusOne(int z)
// ensures: plusOne = #z + 1 and z = #z
```

24

1. To answer Question #2, we must substitute 4 for #z in the ensures clause, #z represents the incoming value of 'z', which was 4.

# Answers: Tracing Table #1

Code	State
<code>y = plusOne(x);</code>	
Question #2: does the client know the following: $y = 5$ ?	$x = 4$ $y = 5$

Second:

- evaluate the ensures clause after the substitution
- The answer is “yes” to Question #2, plusOne returns  $4 + 1 = 5$

```
int plusOne(int z)
// ensures: plusOne = 4 + 1 and z = 4
```



25

1. Here you'll see that we have substituted 4 into the ensures clause giving us the expression stating that  $\text{plusOne} = 4 + 1$ , so plusOne will return the value  $4 + 1$  or 5
2. So the answer to our 2<sup>nd</sup> question is yes, the client knows that the value of the variable 'y' in the client's code will be 5 after plusOne terminates

## Reasoning: Tracing Table #2

<i>Code</i>	<i>State</i>
	x = INT_MAX (e.g., $2^{31} - 1$ ) y = 301
y = plusOne(x);	
	x = ??? y = ???

26

1. Now let's look at a 2<sup>nd</sup> example, but this time the client passes the value INT\_MAX into plusOne – this is shown in the 2<sup>nd</sup> column of the tracing table before the call is made to plusOne

## Question: Tracing Table #2

<i>Code</i>	<i>State</i>
<i>Question #1: did the client satisfy the requires clause?</i>	x = INT_MAX (e.g., $2^{31} - 1$ ) y = 301
y = plusOne(x);	
	x = ??? y = ???

27

1. Question: did the client satisfy the requires clause of plusOne?

## Answer: Tracing Table #2

Code	State
<i>Question: did the client satisfy the requires clause?</i>	$x = \text{INT\_MAX}$ (e.g., $2^{31} - 1$ ) $y = 301$
<code>y = plusOne(x);</code>	

- Substitute the value `INT_MAX` for `z` in the `requires` clause
- We get: `INT_MAX < INT_MAX`
- The answer is “no”, the client did not satisfy the `requires` clause


```
int plusOne(int z)
// requires: z < INT_MAX
```

28

1. Again we substitute value being passed in into the `requires` clause – which is `INT_MAX` in this example.
2. This substitution gives us the Boolean expression `INT_MAX < INT_MAX`, which evaluates to false.
3. So the answer is “no”, the client did not satisfy the `requires` clause.

## Reasoning: Tracing Table #2

Code	State
	x = INT_MAX (e.g., $2^{31} - 1$ ) y = 301
y = plusOne(x);	
	x = INT_MAX y = ???



So we do not know what value will be returned by plusOne  
Or even if it will terminate

29

1. What does this mean to the client?
2. Since the client violated plusOne's contract, plusOne is not bound to the contract, so the client does not know what will be the value of 'y' after the call.
3. The client does not even know if the plusOne will terminate.



# Terminology Wrap-up

design-by-contract  
ensures clause  
formal parameters  
mathematical model  
postcondition

precondition  
requires clause  
return type  
terminate

30

1. Here is a list of some of the terminology talked about in today's slides.
2. Be sure you feel comfortable with all of these terms because we will be using them frequently from now on.

# Main Points

- *Design-by-Contract*
  - Mathematical and informal models
  - Operation contracts
  - Responsibility in Design-by-Contract
  - Reasoning using a contract – one way is with a tracing table

31

1. And there you have it, Design by Contract
2. With Design by Contract we utilize both formal mathematical models and informal models as the basis for writing down the contract that governs the behavior of the operation.
3. In today's example we used formal contracts based on mathematical Integers for the contract of `plusOne`,. Other examples in the future will at times rely on less formal models.
4. With these contracts, we know who is responsible for what and this
5. Along with the contract allows us to precisely reason about the behavior of a program.

# Credits

These slides were adapted from slides obtained from Dr. Bruce W. Weide and Dr. Paolo Bucci.

Drs. Weide & Bucci are members of the Resolve/Reusable Software Research Group (RSRG) which is part of the Software Engineering Group in the Department of Computer Science and Engineering at The Ohio State University.

32

- These slides were adapted from slides obtained from Dr. Bruce W. Weide and Dr. Paolo Bucci.
- Drs. Weide & Bucci are members of the Resolve/Reusable Software Research Group (RSRG), which is part of the Software Engineering Group in the Department of Computer Science and Engineering at The Ohio State University.