

Specification Benefits

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Wednesday, Oct. 20, 2021

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 Recap: Verification vs. Testing
 - ▶ Primitive Types and Abstraction
- 2 String Theory
 - ▶ Empty String
 - ▶ Denoting a Specific String
 - ▶ Concatenation
 - ▶ Length
 - ▶ `Reverse` Function
 - ▶ `Prt_Btwn` Function
 - ▶ `Occurs_Ct` Function

String Theory

- ▶ A string can be thought of as a series of zero or more **entries** of *any* other mathematical type, say, T
 - ▶ We will call this $\text{Str}(T)$
 - ▶ Also denoted in computing foundations as T^*

Example: Stack

- ▶ A **Stack** allows entries to be inserted and removed in **LIFO** (last-in-first-out) order

How to Specify This Interface¹

```
public interface IntegerStack {  
  
    void push(Integer x);  
    Integer pop();  
    int depth();  
    ...  
  
}
```

¹Lecture 6

Abstraction

```
/**  
 * mathematical model Str( $\mathbb{Z}$ )  
 * ...  
 * initialization ensures self = <>  
 */  
public interface IntegerStack {  
  
    ...  
  
    void push(Integer x);  
    Integer pop();  
    int depth();  
  
    ...  
}
```

- ▶ $\mathbb{Z}$ is the mathematical set of integers
- ▶ **initialization ensures** is the specification for the constructor.
 - ▶ Can also be written as: `self = empty_string`

Example

Tracing Table: Need to Be Filled

<i>Code</i>	<i>State</i>
<pre>IntegerStack si = new Stack1();</pre>	

Example

Tracing Table: Filled

<i>Code</i>	<i>State</i>
<code>IntegerStack si = new Stack1();</code>	
	<code>si = <></code>

Push Specification

```
/**
 * mathematical model Str(Z)
 * ...
 * initialization ensures self = <>
 */
public interface IntegerStack {

    ...

    /**
     * @post self = <#x> o #self and x = #x
     */
    void push(Integer x);

    ...
}
```

- ▶ `#self` is the value of this stack before `push`
- ▶ Note that if this is a bounded stack, this interface would have a constant called `MAX_DEPTH`.
 - ▶ Need to add a precondition that says: `|self| < MAX_DEPTH`

Example

Tracing Table: Need to Be Filled

Code	State
	$si = \langle 3, 70 \rangle$ $k = 49$
<code>si.push(k);</code>	

Example

Tracing Table: Filled

Code	State
	$si = \langle 3, 70 \rangle$ $k = 49$
<code>si.push(k);</code>	
	$si = \langle 49, 3, 70 \rangle$ $k = 49$

Pop Specification

```
/**
 * mathematical model Str(Z)
 * ...
 * initialization ensures self = <>
 */
public interface IntegerStack {

    ...

    /**
     * @pre |self| > 0
     * @post #self = <pop> o self
     */
    Integer pop();

    ...
}
```

- `#self` is the value of this stack before `pop`

Example

Tracing Table: Need to Be Filled

Code	State
	$si = \langle 49, 3, 70 \rangle$ $z = -584$
$z = si.pop();$	

Example

Tracing Table: Filled

Code	State
	$si = \langle 49, 3, 70 \rangle$ $z = -584$
<code>z = si.pop();</code>	
	$si = \langle 3, 70 \rangle$ $z = 49$

Alternative Pop Specification

```
/**
 * mathematical model Str(Z)
 * ...
 * initialization ensures self = <>
 */
public interface IntegerStack {

    ...

    /**
     * @pre |self| > 0
     * @post self = Prt_Btwn(1, |#self|, #self) and
     *         <pop> = Prt_Btwn(0, 1, #self)
     */
    Integer pop();

    ...
}
```

- ▶ **#self** is the value of this stack before **pop**

Depth Specification

```
/**
 * mathematical model Str(Z)
 * ...
 * initialization ensures self = <>
 */
public interface IntegerStack {

    ...

    /**
     * @post depth = |self|
     */
    int depth();

    ...
}
```

Inject Specification

- ▶ We want to add a method called `inject` that would take in an integer to add to the stack and a position to add it to.
 - ▶ `void inject(Integer val, int pos);`
- ▶ Calling `inject(7, 3)` would add 7 to the third position in our stack, and shift everything else after that position back
 - ▶ **Note:** Indexing starts at one for `inject`
- ▶ How would we write the contract for this?

Inject Specification

Precondition

- ▶ What is our precondition?

Inject Specification

Precondition

- ▶ What is our precondition?
 - ▶ The position has to exist in the stack
 - ▶ Can't add something to the 7th position if there are only 5 items in the stack
- ▶ How can we write this formally?
 - ▶ Using our string notation

Inject Specification

Precondition

- ▶ What is our precondition?
 - ▶ The position has to exist in the stack
 - ▶ Can't add something to the 7th position if there are only 5 items in the stack
- ▶ How can we write this formally?
 - ▶ Using our string notation
 - ▶ `@pre 0 < pos <= |self| + 1`

Inject Specification

Postcondition

- ▶ What is our postcondition?

Inject Specification

Postcondition

- ▶ What is our postcondition?
 - ▶ `val` will be inserted into the stack at position `pos`, and the rest of the stack will shift down
- ▶ How can we write this formally?

Inject Specification

Postcondition

- ▶ What is our postcondition?
 - ▶ `val` will be inserted into the stack at position `pos`, and the rest of the stack will shift down
- ▶ How can we write this formally?
 - ▶ Use `Prt_Btwn`
 - ▶ Recall, position for `inject` starts at 1
 - ▶ Position for `Prt_Btwn` starts at 0

Inject Specification

Postcondition

- ▶ What is our postcondition?
 - ▶ `val` will be inserted into the stack at position `pos`, and the rest of the stack will shift down
- ▶ How can we write this formally?
 - ▶ Use `Prt_Btwn`
 - ▶ Recall, position for `inject` starts at 1
 - ▶ Position for `Prt_Btwn` starts at 0
 - ▶ `@post self = Prt_Btwn(0, #pos-1, #self) o <#val> o Prt_Btwn(#pos-1, |#self|, #self) and val = #val and pos = #pos`

Code for inject

- ▶ How can you write the code for this
- ▶ Remember, this is a secondary operation
- ▶ Implement it using our primary operations on your own

A Good Specification Is...

- ▶ Abstract
- ▶ Simple
- ▶ Clear
- ▶ Concise
- ▶ Consistent
- ▶ Implementation-neutral
- ▶ Precise
- ▶ Others ...

Understanding the Context

- ▶ Code
- ▶ Contracts
 - ▶ Why do we need them? Because ...
 - ▶ How? Java interfaces + abstraction
- ▶ Formal Contracts
 - ▶ Why do we need them? Because ...
 - ▶ How? Use math in abstraction

Tracing

- ▶ Abstraction helps us with tracing
- ▶ We have a clear mathematical notation for the data type we are working with
 - ▶ Even for things like stacks and queues
- ▶ We can trace through using only that notation
 - ▶ Don't have to worry about implementation details

Example

<i>Code</i>	<i>State</i>
<code>IntegerStack si = new Stack1(100);</code>	
	<code>si = <></code>

Example

Code	State
	$si = \langle 3, 70 \rangle$ $k = 49$
<code>si.push(k);</code>	
	$si = \langle 49, 3, 70 \rangle$ $k = 49$

Recall: Black-Box Testing

- ▶ When test plans are based entirely on contracts, the approach is termed black-box testing (code is treated like a black-box)
 - ▶ Black-box test cases don't change if implementation code changes
- ▶ Our abstraction helps here as well
 - ▶ We don't need to know the implementation details
 - ▶ Our abstract contracts tell us everything we need to know

Recall: Black-Box Testing

Abstraction Benefits (e.g., [push](#))

Inputs	Results	Reason
<code>self = < ></code> <code>x = 1</code>	<code>self = <1></code> <code>x = 1</code>	boundary
<code>self =</code> <code><10, 2, 3, 4></code> <code>x = 8</code>	<code>self =</code> <code><8, 10, 2, 3, 4></code> <code>x = 8</code>	boundary, assuming maxDepth = 5
<code>self = <10></code> <code>x = 8</code>	<code>self = <8, 10></code> <code>x = 8</code>	routine

Benefits for Users in Reasoning

- ▶ With the aid of abstraction, users can reason about their code involving objects much in the same way they reason about code involving integers or other primitive types

Formal Reasoning

- ▶ Remember, we want to be able to reason formally with our code
 - ▶ Tracing with abstract values
 - ▶ Helps us prove our code is correct
- ▶ Our abstraction gives us a way to reason formally about what our code does without knowing specific values

What Does This Code Do?

Code	State
	<i>si = #si, val = #val, pos = #pos</i>
<i>si.inject(val, pos);</i>	

What Does This Code Do?

Answer

Code	State
	<i>si = #si, val = #val, pos = #pos</i>
<i>si.inject(val, pos);</i>	
	<i>si = Prt_Btwn(0, #pos-1, #si) o <#val> o Prt_Btwn(#pos-1, #si , #si)</i>

Does the Code Meet This Specification?

- ▶ For users to depend on just interfaces, interfaces must contain what users need to know and that goes beyond operation names (methods) and parameters
- ▶ This is what contract specification with abstraction gives us

Power of Abstraction

- ▶ Notice we can do the testing, tracing, writing user code and everything without knowing anything about the class implementation(s) of `IntegerStack` interface!

- ▶ Driver's Education and Formal Interface Specifications:
<https://www.cs.clemson.edu/resolve/research/reports/RSRG-11-05.pdf>

Summary

1 IntegerStack

- ▶ Abstraction
- ▶ Push Specification
- ▶ Pop Specification
- ▶ Depth Specification
- ▶ Inject Specification

2 Understanding the Context

3 Tracing and Black-Box Testing

4 Benefits for Users in Reasoning

5 Formal Reasoning

6 Does the Code Meet This Specification

7 Power of Abstraction