LECTURE 9
PROTOCOLS

Outline

- Protocols: what and why?
- Documenting protocols
  - Pre and Post Conditions
- Group exercise
Review

- Responsibility?
- Contract?

Goals

- Refine design
  - Refine classes and responsibilities
  - Associate one or more methods (procedures) with each contract
- Define protocols
  - Protocols
    - set of signatures for methods to be implemented
  - Signatures
    - method name, input parameters, and return type
Why Write Protocols

- Define an interface to a class.
- Why?
  - Delegate implementation to programmers
    - Specialization of personnel (analyst, designer, programmer)
  - Maintenance
    - Program understanding
  - Test Case
    - Derive test cases from pre and post conditions

Protocol Structure

- Signature
  - Method name
  - Type of return value
  - Type of input parameters
  - Description of input, output parameters
- Purpose
  - Pre-condition // require
  - Post-condition // ensure
Two Kinds of Interfaces

- Suppose we have a library class called Math. What do we have to know to use it?
  - Syntactic interface:
    - signature, e.g.,
      ```java
      public static double sqrt(double x) throws IllegalArgumentException;
      ```
  - Semantic interface:
    - behavior, e.g.,
      "If the parameter $x$ is a positive number, the method $\text{sqrt}$ returns a square root approximation of $x$; otherwise it throws an IllegalArgumentException."

Example - Interface Description

```java
/** Returns a square root approximation of the argument $x$. The argument $x$ is assumed to be a positive number. * /

public static double sqrt(double x) { /* … */ }
```

<table>
<thead>
<tr>
<th></th>
<th>Obligations</th>
<th>Rights</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Pass a positive number</td>
<td>Get a square root approximation</td>
</tr>
<tr>
<td>Server (collaborator)</td>
<td>Compute and return a square root</td>
<td>Receive a positive number</td>
</tr>
</tbody>
</table>
Example Revisited

```java
/* Returns a square root approximation of the argument x. */
//@ requires x is positive;
//@ ensures \result is a square root approximation of x;

public static double sqrt(double x) { /* … */ }
```

Definition

- **Pre-condition**
  - A method’s *pre-condition* says what must be true to call it and for the method to function correctly.

- **Post-condition**
  - A method’s *normal post-condition* says what is true when it returns normally (i.e., without throwing an exception).

- A method’s *exceptional post-condition* says what is true when a method throws an exception.
  - `//@ signals (IllegalArgumentException e) x < 0;`
Properties

- All ADTs and objects possess properties (behavioral rules).
- Properties can be stated using a set of Pre- and Post-conditions for each method.
- Properties can be stated using a set of invariant for each object or ADT.
- Good software engineering practices recommend that you state Pre and Post-conditions for all methods.
  - Provide a set of conditions that the implementer of the method must meet
  - Help the reader to know what they can expect of the method

Pre-Condition

- Capture the conditions that must be true in order for the method to execute correctly
- Describe the required state of the ADT or object before entering the function
- Written as an expression that is true or false
- May consist of statements connected by logical operators (AND, OR)
- Use true when no pre-condition exists
Post-Condition

- Must clearly state what is true when the method completes execution
- Describe the expected state upon exiting the function
- Should be strong enough so that only correct implementations will satisfy the condition

Notes

- Generate protocols for main responsibilities
- Protocols to public methods must be unambiguous. Why?
- Protocols to private methods are notes to developer
- Common to discover holes in design at this point
  - If so, repeat earlier phases of design
Process for Writing Protocols

- For each class
  - For each contract
    - For each responsibility
      - Specify complete protocol (set of signatures) to support the responsibility

Specifying Classes - 1

- For each contract within a class, list the responsibilities that support it.

- Under each responsibility, write
  - the signature of the method(s) that implement the responsibility,
  - a brief description and note the collaborations required for the method.

- Add implementation notes, algorithms, constraints (time, space), error conditions.
Specifying Classes - 2

**Class**: Drawing

**Superclasses**: DisplayableObject

**Subclasses**: None

**Class Diagram**: see Figure 2-1

**Collaborations Diagram**: see Figure 3-5

**Description**: Represents the structure of the elements ...

**Contracts**

2. Maintain the elements in a drawing

Know which elements are contained in the drawing

For every contract

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addElement(DrawingElement)</code></td>
<td>Add a drawing element to the front of the list of drawing elements.</td>
</tr>
<tr>
<td><code>uses List</code></td>
<td></td>
</tr>
<tr>
<td><code>Pre</code>: The element is not already contained in the list.</td>
<td></td>
</tr>
<tr>
<td><code>Post</code>: The element is the first element of the list.</td>
<td></td>
</tr>
</tbody>
</table>

Specifying Classes - 3

**elementAt(Point) returns DrawingElement**

**uses List, DrawingElement (3)**

Returns the drawing element at the given point.

**Pre**: none

**Post**: if boundary rectangle contains Point then

- returned is the first such drawing element in List
- else returned is a null object

Maintain the ordering between elements

**elementAfter(DrawingElement) returns DrawingElement**

**uses List**

...
Approaches To Writing Pre- and Post-conditions

- Informal specification
- Formal specification
- Semiformal specification

Informal Specifications

- Natural languages (e.g., English, Spanish, ...)

  - Advantages
    - (Maybe) easy to read and write

  - Drawbacks
    - Ambiguous
    - Verbose
    - Hard to analyze and reason about
    - No machine support (e.g., runtime check)
Formal Specification

/*@ requires x > 0.0;
   @ ensures Math.abs(result – Math.sqrt(x)) <= 0.00001;
   @*/

    public static double sqrt(double x) { /* … */ } 

- Object Constraint Language (OCL), …
- Advantages …
- Drawbacks
  - (Maybe) hard to read and write
  - Some aspects hard to completely formalize, e.g., GUIs.ar

Semiformal Notation

- Write formal specifications if possible; otherwise, write informal specifications.
  - Informal description
  - Boolean expression of Java or C#:
  - Optional extension
**Semiformal Specifications**

- Write formal specifications if possible; otherwise, write informal specifications.

```java
public class Person {
    /*@ requires group != null; @*/
    @ ensures (* \result is true iff this person appears in the array group *);
    @*/
    public boolean isMemberOf(Person[] group) { /* … */ }
    // …
}
```

**Semi-formal Description**

- Some text describing a property
- Treated as a Boolean expression, and
- Thus, allows
  - To escape from formality, and
  - To organize natural language as assertions.

```java
/*@ requires (* x is positive *); @*
@ ensures \result > 0 &&
@ (* \result is an approximation to square root of x *)
@*/
public static double sqrt(double x);
```
Boolean Expressions

- Boolean expressions of C# or Java
  - Boolean operators (&&, ||, !)
  - Conditional expression (?): Conditional expression (?:)
  - Built-in primitive operators (+, -, *, /, <, >, <=, etc)

- Write post conditions for the following:

  ```
  //@ ensures \result == (x > y ? x : y);
  public static int max(int x, int y)
  // Hint: F = C * 1.8 + 32
  //@ ensures \result == x * 1.8 + 32;
  public static double celsiusToFahrenheit(double x)
  ```

Optional Extensions

- Old expression (\old)
  - \old(E) : the value of E in the pre-state

  ```
  public class Counter {
      private int value;

      //@ ensures \result == value;
      public int value() { /* ... */ }

      //@ ensures value == \old(value) + 1;
      public void incr() { /* ... */ }

      //@ ensures value == \old(value) – 1;
      public void decr() { /* ... */ }
  }
  ```
Optional Extensions (2)

- Logical operators

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \rightarrow b)</td>
<td>a implies b</td>
</tr>
<tr>
<td>(a \leftrightarrow b)</td>
<td>b implies a</td>
</tr>
<tr>
<td>(a \iff b)</td>
<td>a iff b</td>
</tr>
</tbody>
</table>

- Quantifiers (\(\forall, \exists\))
  - All CS4311 students are undergraduate students
  - Some CS 5375 students are undergraduate students

Quantifiers

- Quantifiers (\(\forall, \exists\))

```java
//@ requires (* a is sorted *);
public void doSomething(int[] a);

What does "sorted" mean? In ascending or descending order?

//@ requires (* for all i and j such that i and j are valid indices of a,
//@ if i is less than or equal to j,
//@ then a[i] is less than or equal to a[j]);

can you be more formal?

//@ requires (forall int i, j;
//@ 0 <= i && i < a.length && 0 <= j && j <= a.length;
//@ i <= j \rightarrow a[i] <= a[j]);
```
Quantifiers (Cont.)

//@ requires (* x appears in a *):
public void doSomething(int[] a, int x);

What does "appears" mean? How many times?

//@ requires (* for some i such that i is a valid index of a,
//@ a[i] equal to x *);

Can you be more formal?

//@ requires (!exists int i; 0 <= i && i < a.length && a[i] == x);
Exercise

public class Car {
    public int getPrice() { /* … */ }

    /** Returns the cheapest car in the array cars */
    /*@ requires cars != null && cars.length > 0; @
    @ ensures (\exists int i; 0 <= i && i < cars.length; cars[i] == result) @
    @ && (\forall int j; 0 <= j && j < cars.length; @
    @ result.getPrice() <= cars[j].getPrice()); @*/

    public static Car cheapest(Car[] cars) { /* … */ }
}

Hint: State that result appears in cars and is the cheapest.

Group Exercise (Post Conditions)

- Groups 1, 2, 3, 4, 5
  - public void push (Object e) {push to top of stack}
  - public void pop () {pop from bottom of stack}

- Group 6, 7, 8, 9, 10
  - public void push (Object e) {push to bottom of stack}
  - public object pop () {pop from top of stack}
Unbounded Stack

```java
public class Stack {

    //@ ensures elems.length == 0;
    public Stack() { /* ... */ }

    //@ ensures result == (elems.length == 0);
    public boolean isEmpty() { /* ... */ }

    //@ ensures elems.length == \old(elems.length + 1);
    //@ ensures elems[elems.length - 1] == e;
    //@ ensures (forall int i; 0 <= i && i <= elems.length - 1;
    //@     elems[i] == \old(elems[i]));
    public void push(Object e) { /* ... */ }

    Push to end of list
    Change this to a correct implementation of stack push

```
Exercise

Fill in the missing pre and post-conditions.

```java
//@
//@ requires ! isEmpty();
//@ ensures result == elems[elems.length - 1];
public Object top() { /* … */ }

//@ requires isEmpty();
//@ ensures elems.length == \old(elems.length) - 1 &&
//@ (forall int i; 0 <= i && i < elems.length;
//@   elems[i] == \old(elems[i]);
//@)
public void pop() { /* … */ }

Q: Change pop to return the popped element.
```

Exercise: ADT List

- Write the description and pre- and post-conditions for the following methods:
  - List (void);
  - boolean isin (int item);
  - void print (void);
  - void count (void);
  - void print_reverse (void);
  - void insert (int item);
  - void delete(int item);

- What will you change if list is an ordered list? Assume the same signatures.
HW Assignment

Define method signatures and semiformal pre and post conditions for the following:

1. Bubble sort
2. Merge sort
3. Binary Search
4. Inverse of a number (1 over the number)
5. Is-prime
6. Factorial
7. Fibonacci \((F_n = F_{n-1} + F_{n-2}, \ n \geq 2)\)
8. Extra Credit: (Ackerman’s function \(A(m,n)\))
   \[
   A(m,n) = \begin{cases} 
   n+1 & \text{if } m = 0 \\
   A(m - 1, 1) & \text{if } n = 0 \\
   A(m - 1, A(m, n - 1)) & \text{else}
   \end{cases}
   \]

Note: you should first try to come up with formal pre-post conditions and only if that is not possible then come up with semiformal conditions.