CSE 331 Software Design and Implementation

Handout C5: ***Class and Method Specifications***

Contents:

Introduction

This handout describes how to document the specifications of classes and methods. This document focuses on practical issues.

This document uses the `Line` class as an example. Notice that we do not provide fields or method bodies in our example. This document covers specifying the behavior of classes and methods (what they should do), not their implementation (what they actually do and how they do it). Abstraction Functions and Representation Invariants covers how to document a class's implementation.

```
/**
 * This class represents the mathematical concept of a line segment.
 *
 * Specification fields:
 *  @specfield start-point : point  // The starting point of the line.
 *  @specfield end-point   : point  // The ending point of the line.
 *
 * Derived specification fields:
 *  @derivedfield length : real     // length = sqrt((start-point.x - end-point.x)^2 + 
(start-point.y - end-point.y)^2)
 *                                  // The length of the line.
 *
 * Abstract Invariant:
 *  A line's start-point must be different from its end-point.
 */
public class Line {

  ... // Fields not shown.

 /**
  * @requires p != null && ! p.equals(start-point)
  * @modifies this
  * @effects Sets end-point to p
  */
  public void setEndPoint(Point p) {
    ...
  }

  ...

}
```

Because the concepts discussed here are interrelated, this document starts with a short list of definitions before diving into the details.

Abstract Value
: What an instance of a class is supposed to represent. For example, each instance of `Line` represents a given line.

Abstract State
: The information that defines the abstract value. For example, each abstract line has a starting point and an ending point.

Specification Fields
: Describes components of the abstract state of a class. For example, the abstract state of a `Line` is made up by the specification fields `start-point` and `end-point`.

Derived Specification Fields
: Information that can be derived from specification fields but is useful to give a name to. For example, `Line` has the derived field `length` which describes the length of the line.

Abstract Invariant
: A condition that must be true over the abstract state of all instances of a class. For example, `Line` requires that no line may have the same starting and end point. Abstract invariants are over a class's abstract state.

Method Specifications
: Describe a method's behaviors in terms of abstract state. For example, `Line`'s `setEndPoint` method reassigns the `end-point` specification field.

The above concepts are included in a class's external specification (in Javadoc). They aid a client with using the class.

The rest of this document is organized as follows. First, it explains how to document what a class abstractly represents using abstract state, specification fields and derived fields. Then, it explains how to specify method behavior, in terms of abstract state.

Abstract Values and Abstract State

To a computer, an instance of an object is a series of zeros and ones. To humans, on the other hand, these zeros and ones represent higher-level abstract ideas: integers, strings, lines, and lists, for example. The abstract value of an object is what the object represents to humans on this abstract level.

The abstract state of an object is what information a human associates with the abstract value of an object. An abstract value is determined by the abstract state. For example, for the `String` `"cat"`, humans associate the following sequence of letters: 'c', 'a', and 't'. This sequence of letters is the abstract state of the `String`.

It is important to note that how the `String` `"cat"` is represented concretely (within the Java code) does not affect its abstract state. Whether `String` internally uses an `array`, a `List`, or some clever compressed numerical encoding for the the characters `'c'`, `'a'`, and `'t'`, the abstract state of `"cat"` is still a just a sequence of those letters.

Mathematical Abstract Values

For some data types (classes), the abstract value space consists of familiar mathematical objects:

- a *set* of integers
- a *sequence* of characters (i.e., a string)
- a *pair* of real numbers (or a triple, or in general a *tuple*)

If so, you're in luck. You can use conventional notation for explaining the class's abstract values and functions on its abstract value (see later sections).

- *set comprehension*: **{ x | P(x) }** denotes the set of all elements *x* that satisfy the property *P*. More generally, **{ f(x) | P(x) }** denotes the set of values of the expression *f(x)* for all *x* that satisfy the property *P*.
- *set union*: **x ? y** denotes the union of two sets *x* and *y*. (This can also be written x + y when there's no danger of confusion with addition.)
- *set membership*: **a ? x** or **a in x** tests whether *a* is an element of the set *x*.
- *sequence construction:* **[a, b, c]** denotes a sequence of three elements.
- *sequence concatenation*: **x : y** denotes the concatenation of two sequences *x* and *y*. (This can also be written **x ^ y** or **x + y** when there's no danger of confusion with exponentiation or addition.)
- *sequence indexing*: **x[i]** denotes the *i*th element of a sequence *x*.
- *set or sequence size*: **|x|** denotes the number of elements in a set or sequence *x*.
- *tuple construction*: **<a, b, c>** is a tuple of three elements. This is also written **(a, b, c)**. Unlike sequences, tuples are fixed-length, so we don't normally think about concatenating them.

You aren't obliged to use this syntax. Some of it is more standard than the rest: everybody with some training in mathematics recognizes set comprehension syntax, but sequence concatenation isn't particularly standardized. You may find it clearer to write sequence concatenation as a function like **concat(x, y)**. What really matters is clarity and lack of ambiguity, so if you have any doubt whether your reader will understand you, just define it: "...where **concat(x,y)** is the concatenation of two sequences **x** and **y**."

Specification Fields

More frequently, the abstract values are not simple mathematical objects. In these cases, it's more useful to think about the abstract value as if it were an object with fields. For example, a line has a *start* and an *end*; a mailing address has a *number*, *street*, *city*, and *zipcode*; and a URL has a *protocol*, *host name*, and *resource name*.

Mathematically, this is the same as a *tuple*; it's just a tuple whose parts are named, rather than simply numbered. So we could specify the abstraction function as a single function that maps representation objects into tuples. But it's convenient, and more readable, to break the abstraction state into several separate parts, where each part is a *specification field*. (Specification fields are more commonly called *abstract fields*, because they're fields of the abstract value, as opposed to *rep fields* which are fields of the representation value. Unfortunately, *abstract* has another meaning in Java, so we will avoid that potentially confusing terminology.)

Specification fields often (but not always) correspond to observers or getters on the abstract data type. Because the structure of abstract values is a matter of interest to the clients of your class, the specification fields should be listed in the class overview. In CSE 331, we have a Javadoc convention for describing them: `@specfield` *`name`* `:` *`type`* `//` *`description`*. Here's an example:

```
/**
 * Represents an appointment for a meeting.
 * @specfield date : Date          // The time
 * @specfield room : integer       // The room number of the meeting's location
 * @specfield with : Set<Person>  // Whom the appointment is with
 */
 class Meeting {
```

By convention, in specification fields, lowercase types like *sequence* or *set* refer to mathematical entities. Capitalized types refer to other ADTs (i.e., Java classes). Where you have a choice, prefer a mathematical entity as the type of a spec field; it is better to use sequence than List, for example. It's more elegant, and reduces the coupling between your specification and concrete Java classes.

The presence of a specification field does not imply anything about the interface or implementation of the class. Although spec fields often correspond to observer methods, that's not always true. (An observer method is one that computes a value without performing any side effects. All getter methods are observers, but not all observers are getters.) The interface might not provide any observers that query the spec field's state, so clients of the class might not have access to the information stored in the spec field. (An example is that a stack implementation might have a spec field for the elements of the stack, but a client might only be able to push and pop rather than being able to obtain the full state of the stack.) Likewise, the implementation might not actually have a concrete field of the spec field's type: that information may be computed from multiple concrete fields, or it might not be available at all. The point is that specification fields are a convenience for use in method specifications and abstraction functions.

Derived Fields

Derived fields are information that can be derived from the specification fields and is useful to give a name to. For example, take the class below:

```
/**
 * Represents a square.
 * @specfield length : int // The length of the square's sides.
 * @derivedfield area : int // area = length^2. The area of the square.
 *
 * Abstract Invariant:
 *  length > 0.
 */
class Square {...}
```

The derived field `area` can be derived by squaring the `length` specification field. A derived field's documentation should state how it is derived from the specification fields.

A derived field's purpose is to help with writing method specifications, abstraction functions, and representation invariants: It is easier to write and understand `area` than `length^2`. Since derived fields can be derived from the specification fields, the abstraction function does not need to relate an object's concrete representation to derived fields. (Derived fields should be explained in terms of only specification fields, not representation fields.) Additionally, method specifications do not need to state a method's effects on a derived fields. For example:

```
/**
 * Represents a square.
 * @specfield length : int // The length of the square's sides.
 * @derivedfield area : int // area = length^2. The area of the square.
 *
 * Abstract Invariant:
 *  length > 0.
 */
class Square {

   /** The length of the square's sides. */
   int size;

   // Abstraction Function:
   //  AF(r) = a square, s, with s.length = r.size.
```

```
  //
  // Representation Invariant:
  //   size > 0

  /** Creates a new Square with length = len.
   * @requires len > 0.
   * @return a new Square, s, with s.length = len.
   */
  public Square(int len) {
    if (len <= 0) throw new IllegalArgumentException(len + " < 0");
    this.size = len;
  }

  /** Returns the difference in area between this and s.
   * @return this.area - s.area.
   */
  public int differenceInArea(Square s) {
    return (this.size*this.size) - (s.size*s.size);
  }

  /** Sets this.length to len.
   * @requires len > 0.
   * @effects sets this.length to len.
   */
   public void setLength(int len) {
     if (len <= 0) throw new IllegalArgumentException(len + " < 0");
     size = len;
   }
}
```

Notice how the method specification of `differenceInArea` uses the derived field `area` to make it easier to explain what it returns.

It is never necessary for a method specification to indicate its effect on a derived specification field because the class documentation has defined the derived specification field in terms of specification fields. Since `area` is a derived field, the constructor does not need to say what the newly constructed `Square`'s `area` is. Similarly, the method specification for `setLength` does not need to document its effect on `area`. See Abstraction Functions and Method Specifications for further details.

Note that one could have made `area` a specification field instead of a derived specification field. This would relieve the programmer from the responsibility of documenting how `area` can be derived from the specification fields. However, in this case, the constructor and `setLength` would be required to specify their effects on `area`.

Suppose you have a derived specification field `f`. It is permissible for there to be a concrete field in the implementation that stores the value of `f`, or for there to be a method that computes the value of `f`, or for there to be no such field or method. That is an implementation detail that is of no interest to anyone who is reading the specification.

Another way to express the difference between a derived field and a specification field is the following. A derived field can be determined from one or more **specification** fields. A specification field *might* be computed from one or more **concrete** fields (or not stored in any concrete fields at all).

Method Specifications

Method specifications describe the behavior of a method in terms of its preconditions and postconditions. Note that method specifications may only refer to specification fields, method arguments, and global variables (aka public static fields), never to the concrete fields of the implementation.

Preconditions

Preconditions are properties that must be true when the method is called. It is the responsibility of the caller to guarantee that these properties hold. If the preconditions do not hold, the method is allowed to behave in absolutely any fashion, including crashing the program, continuing with incorrect results, informing the user of the problem, or gracefully recovering from the problem. Callers should always assume that preconditions are not checked by a method. However, it is good practice - though not required - for an implementation to check its preconditions (if the check can be performed efficiently) and throw an error if they are not satisfied.

Preconditions are indicated by the "**requires**" clause in a method specification. If a "requires" clause is omitted from the method specification, it is assumed that the method does not have any preconditions.

requires (default: no constraints)
: The preconditions that must be met by the method's caller.

Postconditions

Postconditions are properties that a method guarantees will hold when the method exits. However, if the precondition did not hold when the method was called, then all bets are off, and the method may behave in any fashion whatsoever. In particular, if the precondition does not hold upon method entry, then the postcondition need not hold on method exit.

A postcondition can be written as a single complex logical formula, but it is convenient to separate it into logically distinct parts. CSE 331 uses "return", "effects", "throws", and "modifies". (In the descriptions below, "default" indicates what is assumed if that clause is omitted from the method specification.)

return (default: no constraint on what is returned)
: The value returned by the method, if any.

throws (default: none, which means that no exceptions are ever thrown)
: The exceptions that may be raised, and under which conditions. Those conditions must be a subset of the preconditions (the requires clause). The specification should not make representations about the behavior when the preconditions are not satisfied.

modifies (default: nothing, which means that there are no side effects)
: Variables whose value *may* be modified by the procedure. They are not guaranteed to be modified, unless otherwise indicated by he effects clause. If object `x` has specification fields `f`, `g`, and `h`, then "`modifies x`" means that any combination of `x.f`, `x.g`, and `x.h` might be modified. "`modifies x.g, x.h`" would be more restrictive. Often, programmers are more interested in quantities that are *not* listed in the modifies clause, since those are guaranteed not to be changed by the method.

effects (default: true, which means "can have any effect" on the references listed under modifies)
: The side effects of the method, such as changes to the state of the current object, to parameters, or to objects held in global variables. If a specification field is listed in the modifies clause but not in the effects clause, then it may take on any value allowed by the abstract invariants of this class of objects. The difference between the modifies and effects causes is that modifies lists everything that may change and effects indicates what changes occur.

Using Spec Fields for Specifications

Specification fields are useful for writing specifications of the ADT's operations. Here's a specification for a method on the `Line` class:

```
/**
 * @requires l.start-point is equal to this.end-point && l != null.
 * @return a line segment that is equal to this + l; that is, l appended to this.
 */
public Line add(Line l);
```

Specifications may refer to specification fields (such as `start-point` and `end-point`), but *never* to representation fields (such as `start` and `end`). Rep fields depend on a particular implementation, so we don't want to expose them in a specification, which can be implemented in many different ways.

Using Derived Spec Fields for Specifications

When you're writing specifications for operations, you may find it useful to define *derived* spec fields. A derived spec field is just a spec field that can be written in terms of other spec fields. In other words, it's a shorthand.

You can freely use derived spec fields in a method specification (and easing such specifications is the point of derived spec fields):

```
/**
 * @return true is this.length > l.length.
 */
public boolean longer(Line l);
```

Subclasses and overridden methods

A subclass often has a different (stronger) specification than its superclass, and often has a larger abstract state than its superclass. When the specification and abstract state are identical to those of the parent (for instance, an implementation of an abstract class), then there is no need to repeat them in the subclass. However, it is helpful to include a brief note indicating that the superclass documentation should be used instead. That note helps readers to distinguish whether the specification is the same, or the author simply didn't document the class.

When the specifications differ, then you have two options. The first option is to repeat, in the subclass, the full superclass documentation. The advantage is that everything is in one place, which may improve understanding. The second option is to augment the existing specification -- for example, to add a few new specification fields and constraints on them. Whichever you do, make sure that you clearly indicate your approach, and that you are consistent throughout the codebase.

Similar rules hold for a method that overrides another method. It's acceptable to leave the Javadoc blank if the specification is identical. (The generated HTML will use the overridden method's Javadoc documentation, but a normal Java comment is a good hint to someone who is reading the source code.) Otherwise, it is usually better to give the complete specification. If you merely augment the overridden method's specification, be sure to refer to it in the documentation.

Back to the CSE 331 home page.

Back to Conceptual Info
Back to the CSE 331 home page.