TEACHING CLEANROOM SOFTWARE ENGINEERING WITH OBJECT-ORIENTED DATA ABSTRACTION

Gabriel J. Ferrer
Department of Mathematics and Computer Science
Hendrix College
Conway, AR 72032
(501) 450-3879
ferrer@grendel.hendrix.edu

ABSTRACT

This paper describes the benefit of incorporating two ideas from Design-by-Contract [3] into the teaching of Cleanroom Software Engineering [4][5], namely the specification of method behavior in terms of inspector methods, and inheritance of method specifications. The inspectors serve as a specification of the transformation between the abstract data type represented by a class and its internal data representation. By inheriting method specifications, the behavior of augmented polymorphic methods can be specified. Pedagogical examples and student code are presented and discussed.

INTRODUCTION

Existing materials for teaching Cleanroom Software Engineering [4] [5] arguably fail to adequately describe how to reconcile the methodology with object-oriented data abstraction and inheritance. Incorporating two key concepts from Design-by-Contract [3] is most helpful in this regard. These key concepts are the specification of method behavior in terms of inspector methods, and the inheritance of method specifications.

The paper begins with an overview of the relevant ideas from Design-by-Contract. Next, the concept of an intended function is described, followed by a discussion of how to combine these ideas. Pedagogical examples and student-written code are included. This approach is then compared with how the topic is addressed by two textbooks [4] [5].

DESIGN-BY-CONTRACT

The Design-by-Contract concept [3] is based on specifying the behavior of each method by using assertions. These assertions take the form of preconditions, postconditions, and invariants. Preconditions and postconditions are analogous to their namesakes in Hoare triples [1]. Each method in a class has a list of preconditions that must be true before it is invoked, and a list of postconditions that will be true after it terminates. Furthermore, a class may have invariants that are postconditions of the constructors and both preconditions and postconditions for all other methods.

All of these assertions are expressed in terms of inspectors; that is, methods that describe a state element of an object's data abstraction without modifying the object.

Each class needs sufficient inspectors to fully describe the abstract state of an object.

Preconditions, postconditions, and invariants are all inherited. That is, any polymorphic method must conform to the assertions in its parent class, and non-polymorphic methods must still correspond to the invariants. A polymorphic method may disjoin the inherited precondition or conjoin the postcondition with additional assertions.

INTENDED FUNCTIONS

In Cleanroom Software Engineering, code segment specifications are expressed using intended functions [4] [5], a variation on the concept of a Hoare triple [1]. Instead of using assertional postconditions, an intended function describes the effect of a code segment as a mapping of the input state of the subprogram onto an output state.

Intended functions are equivalent to Hoare triples in terms of expressiveness. The difference in practice is that intended functions facilitate proofs that use algebraic substitution, manipulation, and simplification. Because a proof technique is the primary means of finding bugs using Cleanroom, this distinction is of great importance.

Every intended function is a concurrent assignment, in that from the point of view of the function, all of the variables change value simultaneously. The values on the right-hand side of the concurrent assignment are the values the variables had prior to the assignment. For example, the intended function [x, y := y, x] represents a swap operation. A segment of code annotated with intended functions appears below:

```
// [y > 0 -> quotient, rem := x div y, x mod y]
quot = 0; rem = x;
// [y > 0 -> quot, rem := quot + rem div y, rem mod y]
while (rem >= y) {
  quot = quot + 1; rem = rem - y;
}
```

The first intended function specifies the overall behavior of the code block; the second function specifies the behavior of the loop. Both intended functions have $y > 0$ as a precondition. The verification process proceeds as follows. First, we observe that by substituting the assigned initial values for quot and rem into the intended function for the loop, we get [y > 0 -> quot, rem := 0 + x div y, x mod y], yielding the overall function.

Next, we must verify the loop. First, we see that the loop terminates because, using our precondition, rem must decrease on each iteration while y stays the same. The next two steps of the verification employ mathematical induction. We begin by showing that if the loop condition is false, that the intended function is trivially true. In this case, if rem < y, then rem div y is 0 and the quot part of the intended function checks out. Similarly, by definition if rem < y and rem is unchanged, then it is equal to rem mod y.

Finally, we show that if the condition is true, then if composing the loop body with the loop intended function is equivalent to that intended function, then the loop is correct. In this case, we get this intended function [quot, rem := (quot + 1) + (rem - y) div y, (rem - y) mod y], which can be transformed further into [quot, rem := quot + y div y + rem div y - y div y, rem mod y - y mod y]. After simplifying, we get the original intended function for the loop. At that point, the code has been verified by using algebraic substitution, manipulation, and simplification.

INCORPORATING DESIGN-BY-CONTRACT INTO INTENDED FUNCTIONS

Existing pedagogy regarding intended functions can benefit from specifying method behavior in terms of inspector methods and inheriting method specifications. In our earlier example, all state changes were described in terms of primitive variable types. Most examples in the two primary Cleanroom textbooks are of this flavor [4] [5]. In the example that follows, inspectors are used to specify the result of an intended function.

```
public class Circle {
   private int x, y, diameter;

   // Inspectors
   public int getRadius() {return diameter / 2;}
   public int getXCenter() {return x + getRadius();}
   public int getYCenter() {return y + getRadius();}

   // [getXCenter(), getYCenter() := getXCenter() + xChange, getYCenter() + yChange]
   public void move(int xChange, int yChange) {
      x += xChange;  y += yChange;
   }
}
```

The external abstraction presented is in terms of the coordinates of the center, but the internal representation is in terms of the upper-left corner of the circle's bounding box. Therefore, the intended function is stated for the move() method in terms of the public methods, even though it is private data members that are being modified.

To verify the method against the intended function, we begin by determining the values that the private data members will hold when the method completes. Then, we check to see if the mappings defined by the inspector methods yield the results specified by the method's intended function. This avoids the overhead of creating an abstraction function [5] while preserving the ability to be fully rigorous in the verification.

Verification of the method move() proceeds as follows. The variable x becomes x + xChange, and y becomes y + yChange. From the intended function, we see that getXCenter() is transformed into getXCenter() + xChange and getYCenter() is transformed into getYCenter() + yChange. The method getXCenter() becomes x + xChange + getRadius(), and by commutativity of addition it becomes (x + getRadius()) + xChange. By substituting, that expression becomes getXCenter() + xChange. Using this reasoning with getYCenter() as well, we can say that the move() method is verified.

The next example, consisting of an interface and an implementing class, shows how intended functions can be inherited. The intended function from the base class is augmented with additional assignments in the child class. These must not disrupt the base class method specification if the child class method is to be considered correct.

```
public interface TicTacToePlayer {
   // [board.isLegal(result) := true]
   public int getNextMove(TicTacToeBoard board);
```

```
}

public class SlightlySmartPlayer implements TicTacToePlayer {
   // [board.isLegal(result), result := true,  a number indicating a square that will yield a
   // win if such a square exists; otherwise, a random number between 1 and 9]
   public int getNextMove(TicTacToeBoard board);
}
```

CLASSROOM EXPERIENCE

The five students in the most recent offering of this course, working as a team, implemented a web browser as their final project, using the Java programming language. They wrote 23 distinct classes for this project, with 14 of these classes inheriting from a parent, and eight of these classes serving as parents in an inheritance hierarchy. The code excerpts in this section are all taken from the code these students wrote for that project.

The class excerpt below illustrates the use of inspectors in specifying method behavior. The return value is defined in terms of the inspector method whoContains(x, y). The use of this inspector masks the reference to the private instance variable document.

```
public class HTMLCanvas {
   private HTMLDocument document;

   public HTMLCompnent whoContains(int x, int y) {
      if (document == null) {return null;}
      return document.whoContains(x, y);
   }

   /* [ whoContains(x, y) = null -> result := null
      | TRUE -> result := whoContains(x, y).getHREF() ] */
   public String getHREF( int x, int y ) {
      HTMLComponent component = whoContains(x, y);
      if( component == null ) {return null; }
      return component.getHREF();
   }
}
```

The class excerpt below shows an example of inheritance. The intended function for the implementation is compatible with its parent in a way that utilizes the inspector methods. The parent indicates that whatever the child does should be within the bounds of x and x.getWidth(). In the subclass, the intended function explicitly describes how this constraint is met by drawing the horizontal line with length getWidth().

```
public abstract class HTMLComponent {
   /* [ page := page plus a rendering of this object that alters only the area between (x, y)
        and (x + getWidth(), y + getHeight()) ] */
   protected abstract void renderOffset(Graphics2D page, int x, int y);
}
```

```
public class HTMLHorizontalRule extends HTMLComponent {
   // A representative implementation of the method
   /* [ page := page plus a black horizontal line of length  getWidth(), vertically centered
        in this object's bounding box ] */
   protected void renderOffset(Graphics2D page, int x, int y) {
      page.setColor( Color.BLACK );
      page.drawLine(x, y + RULE_HEIGHT /2, x + getWidth(), y + RULE_HEIGHT / 2);
   }
}
```

An issue that must be addressed carefully is that there is an intrinsic tension between an informal intended function that is "good enough" and the more formal variety. It is important to avoid being too dogmatic about the use of inspectors to describe object state as long as the resulting intended function is unambiguous. A legacy class will often fail to provide inspectors that deliver the desired state information. The example of renderOffset() shown above demonstrates this. It would be awkward to devise a means of describing a change to a Graphics2D object with inspectors. It is sufficiently unambiguous to indicate in plain English the material that will be drawn on the object.

The most significant problem encountered was that students would often neglect to write inspector methods for private data members that their other methods modified. Consequently, the intended functions for the methods would often reference private variables. The students in the course did not consistently perceive a beneficial tradeoff between the effort to create the inspectors and the benefits of proper data encapsulation.

The excerpt below concisely demonstrates this problem. First, the getFile() method is a private inspector method. Since getResource() is a public method, the method getFile() needs to be public as well in order to be an appropriate specification. Second, download and timeout are both private static variables. The student should either have made them public and final or should have created inspector methods to access them.

```
public class Download {
   /*[ getFile(s,true) = null -> result := null | getFile(s,true) = download -> result := null
     | getFile(s,true) = timeout -> result := null | TRUE -> result := getFile(s,true)] */
   public static String getResource(String s){
      String temp = getFile(s,true);
      if(temp == null || temp.equals(download) || temp.equals(timeout)) {return null;}
      else {return temp; }
}
```

Another alternative is to write the intended function less formally as follows:
[getFile(s, true) is a legal resource -> result := getFile(s, true) | TRUE -> result := null]
In this latter version, rigorous but informal natural language is sufficient to specify the method's behavior concisely, along the lines of the specification of renderOffset() shown in an earlier example. Future lectures will make the option of this tradeoff more explicit.

EXISTING PEDAGOGY

Stavely's textbook [5] is an introduction to Cleanroom Software Engineering techniques that places a heavy emphasis on the use of intended functions. The principal specification technique employed in the context of data abstraction is Hoare's concept of an abstraction function [2], which Stavely calls a "meaning function." This function maps the implementation of an abstract data type to the externally visible abstraction. Two examples of programs with meaning functions are included. In the first example, no inspector methods are given; hence, the intended functions cannot be described by them. The second example includes an abstract base class and two child classes. The intended functions from the base class are inherited unchanged into the child classes. The pedagogical example shown earlier goes beyond these because of the use of inspector methods within the intended functions themselves, and because of its augmentation of the intended functions of the child classes in a manner compatible with the base class.

The book Cleanroom Software Engineering: Technology and Process [4] contains an extended example of the application of the methodology. The aforementioned extended example is a satellite control system, coded in Java. The example makes use of inheritance through interfaces, but intended functions are not used to describe the polymorphic behavior of the methods. Other than the use of inheritance manifest in their examples, the authors do not discuss the implications of inheritance for the construction of intended functions anywhere in the textbook. Furthermore, the intended functions that are used violate encapsulation in several places. The writing of intended functions is not a major emphasis of [4]; readers are encouraged to consult [5] for details.

CONCLUSIONS

For students to learn Cleanroom Software Engineering, they must master incorporating intended functions into their code. The contribution of this paper is the description of how to improve the teaching of intended functions in an object-oriented programming language by incorporating from Design-by-Contract [3] specification of method behavior in terms of inspectors, and inheritance of method specifications.

The resulting benefits include the ability to avoid writing an explicit specification of the mapping between public and private data, and the ability to reason about augmented behavior of polymorphic methods. Pedagogical examples have been given that demonstrate these ideas, and excerpts from student code show results of employing this pedagogy. A pitfall encountered when teaching students to specify method behavior in this manner has been described, and a means for addressing this pitfall was discussed.

This approach has been contrasted with the pedagogy contained in two important textbooks. As [5] is presently being used in the author's course, the approach described here is complementary to the otherwise outstanding pedagogy that this book represents.

The next step of this project is to develop pedagogy to show students how to apply Meyer's concept of an exception [3] in an intended function. Existing Cleanroom pedagogy does not address exceptions explicitly. The author is in the process of developing some verification questions for use with programs containing exceptions that are compatible with the theoretical framework in which the Cleanroom method operates.

REFERENCES

[1] Hoare, C. A. R., An axiomatic basis for computer programming, *Communications of*

the ACM, 12(10):576–583, October 1969.
[2] Hoare, C. A. R., Proof of correctness of data representations, *Acta Informatica*, 1:271–281, 1972.
[3] Meyer, B., *Object-Oriented Software Construction, 2nd Edition,* Prentice Hall, 1997.
[4] Prowell, S.J., Trammell, C.J., Linger, R.C., Poore, J.H., *Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999.
[5] Stavely, A.M., *Toward Zero-Defect Programming*, Addison-Wesley, 1999.