Studying Program Correctness by Constructing Contracts

Timothy S. Gegg-Harrison[1], Gary R. Bunce[2]
Rebecca D. Ganetzky[1], Christina M. Olson[2], Joshua D. Wilson[2]

[1]Department of Computer Science
Oberlin College
Oberlin, OH 44074 USA
tsg@cs.oberlin.edu

[2]Department of Computer Science
Winona State University
Winona, MN 55987 USA
gbunce@winona.edu

Abstract

Because the concept of program correctness is generally taught as an activity independent of the programming process, most introductory computer science (CS) students perceive it as unnecessary and even irrelevant. The concept of contracts, on the other hand, is generally taught as an integral part of the programming process. As such, most introductory CS students have little difficulty understanding the need to establish contracts via preconditions and postconditions. In order to improve teaching program correctness concepts, we implemented ProVIDE, an enhanced integrated development environment (IDE) for Java [7]. ProVIDE supports a modified version of the "design by contract" methodology [13] that assists its student programmers in contract construction. Rather than asking for both a precondition and postcondition for each of his/her methods, ProVIDE asks the student to simply supply a postcondition. ProVIDE then helps the student construct the appropriate precondition by leading him/her through an axiomatic proof of the correctness of the method. Thus, the proof of correctness of the method is a side-effect of the student's need to construct an appropriate precondition.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/program verification – *programming by contract, correctness proofs, formal methods.*
F.3.1 [**Logics and Meanings of Programs**]: Specifying, verifying, and reasoning about programs – *assertions, invariants, pre- and post-conditions.*

General Terms

Documentation, Verification.

Keywords

Design by contract, axiomatic semantics, Java.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ITiCSE '03, June 30 – July 2, 2003, Thessaloniki, Greece.

Copyright 2003 ACM 1-58113-672-2/03/0006…$5.00.

1 Introduction

Due to the dependency of computing on discrete mathematics, the Computing Curricula 2001 (CC2001) Task Force proposed that *discrete structures* be added as a separate knowledge area [10]. Although computer scientists understand the importance of discrete mathematics to the foundations of their field, computer science (CS) students do not always see the relevance. Our experience in teaching computer science over the past several years has shown that incoming CS students are deficient in mathematics in general and they do not see the relevance of mathematics to computer science. Although our CS students were taking discrete mathematics during their first year, they were apparently not retaining it.

In order to address this problem, we recently restructured our CS curriculum. In this restructured curriculum, CS students take two semesters of discrete mathematics, a Discrete Mathematics course that is taught by the Mathematics Department followed by what CC2001 refers to as CS110 (Discrete Structures) that is taught by the Computer Science Department, while they are taking CS101O and CS102O, CC2001's objects-first model for the introductory computer science curriculum. The additional semester of discrete mathematics has helped, however, we believe that the current success of CS students in CS101O and CS102O has significantly benefited from a very active attempt to make discrete mathematics more relevant. We have attempted to show our students this relevance by integrating discrete mathematics via CS-Complete examples, unifying examples that are applicable in CS101O, CS102O, and Discrete Mathematics [6].

In order to better integrate discrete mathematics into the introductory CS curriculum, we implemented ProVIDE, an enhanced integrated development environment (IDE) for Java that enables students to analyze their computer programs (in terms of their correctness) while they are creating them [7]. The primary goal of the construction of ProVIDE is the seamless integration of analysis with the creation of computer programs. Because the concept of program correctness is generally taught as an activity independent of the programming process, most introductory CS students perceive it as unnecessary and even irrelevant. The concept of contracts, on the other hand, is generally taught as an integral part of the programming process. As such, most introductory CS students have little difficulty understanding the need to establish contracts via preconditions and postconditions. The approach we have taken with ProVIDE is a modified version of the "design by contract" methodology [13]. Rather than asking

the student for both a precondition and postcondition for each of his/her methods, ProVIDE asks the student to simply supply a postcondition. ProVIDE then helps the student construct the appropriate precondition by leading him/her through an axiomatic proof of the correctness of the method. Thus, the proof of correctness of the method is a side-effect of the student's need to construct an appropriate precondition.

In the next section, we consider the process of contract construction. We begin by defining the semantics of the Java programming language using axioms and rules, with the assumption that the Java code is free of side-effects. After defining an axiomatic semantics for Java, we consider a sample session with ProVIDE. We highlight how ProVIDE helps the student construct preconditions for methods containing a combination of simple assignment statements, a conditional statement, and a loop. We also show how ProVIDE assists the student in the construction of loop invariants. Conclusions and directions for future research are given in the last section.

2 Constructing Contracts in ProVIDE

ProVIDE was developed by extending Netbeans, a modular standards-based open source integrated development environment written in Java. ProVIDE uses iContract's Javadoc tags **`@pre`**, **`@post`**, and **`@invariant`** [11] that correspond to Eiffel's assertion constructs **`require`**, **`ensure`**, and **`invariant`** [12] for preconditions, postconditions, and invariants, respectively. After students have constructed and debugged their methods, ProVIDE helps them construct assertions for the **`@pre`** tags by guiding them through proofs of correctness using axiomatic semantics [4,5,8,9] to find weakest preconditions, starting with the assertions given in the **`@post`** tags and ultimately generating **@pre** tags that contain the constructed precondition assertions.

2.1 Contract Axioms and Rules

Traditionally, axiomatic semantics have been used to prove that a program segment is correct with respect to its precondition and postcondition. The semantics of an imperative programming language are defined by the Assignment Axiom and a set of inference rules for each of the language's control structures. Although the Assignment Axiom and Sequence Rule are constructive (i.e., they define how to construct the precondition from the postcondition), the inference rules that are generally given for conditional statements and loops are non-constructive (i.e., they assume both a precondition and postcondition have been provided).

ProVIDE's goal is help its student programmers *construct* appropriate preconditions given postconditions that they provide. As such, we want to define a constructive axiomatic semantics for Java using contract-enriched Hoare triples of form $\{P\}\,C\,\{Q\}$ that states that P is the (constructed) weakest precondition of the block of Java code C given the postcondition Q. We use rules of inference of the form $\frac{H_1, H_2, \ldots, H_n}{H}$ that states that the precondition of the contract-enriched statement H is appropriate if we construct appropriate weakest preconditions for $H_1, H_2, \ldots, H_n$.

The **Assignment Axiom** is the fundamental axiom for imperative programming languages, infusing the notion of program state change into our logic.

$$\{P[V \to E]\}\ V = E; \{P\} \qquad \textbf{[Assignment Axiom]}$$

The **Assignment Axiom** defines the weakest precondition of an assignment statement as the postcondition with all the occurrences of the variable on the left hand side of the assignment statement replaced with the expression on the right hand side of the assignment statement. The **Assignment Axiom** also applies to **`return`** statements where the pseudo-variable *@return* (which is used in postconditions to denote the value of the method) replaces V and the expression that is returned replaces E.

The weakest precondition of a block of statements can be constructed by successively backing over each of the statements in the block (starting with the last statement) and using the constructed weakest precondition of each statement as the postcondition to its preceding statement. The following **Sequence Rule** defines the precondition construction process for a pair of statements. Repeated applications of the **Sequence Rule** can be used for blocks containing more than two statements.

$$\frac{\{P\}\,C_1\,\{R\}, \{R\}\,C_2\,\{Q\}}{\{P\}\,C_1; C_2; \{Q\}} \qquad \textbf{[Sequence Rule]}$$

The weakest precondition of a conditional statement can be constructed by considering both possible paths through the **`if-else`** statement. The constructed precondition is the disjunction of a pair of conjuncts that are formed from the guard conjoined with the weakest precondition constructed by backing over the then-clause of the conditional disjoined with the negation of the guard conjoined with the weakest precondition constructed by backing over the else-clause of the conditional.

$$\frac{\{P_1\}\,C_1\,\{Q\}, \{P_2\}\,C_2\,\{Q\}}{\{(P_1 \wedge B) \vee (P_2 \wedge \neg B)\}\ \mathtt{if}\,(B)\,C_1\ \mathtt{else}\ C_2; \{Q\}} \qquad \textbf{[Conditional Rule]}$$

The **Conditional Rule** is used to construct the precondition $(P_1 \wedge B) \vee (P_2 \wedge \neg B)$ where P_1 is constructed from $\{P_1\}\,C_1\,\{Q\}$ and P_2 is constructed from $\{P_2\}\,C_2\,\{Q\}$. Note that since the weakest precondition to the "null" C_2 statement is simply the postcondition Q when the else-clause is omitted then the **Conditional Rule** is used to construct the precondition $(P_1 \wedge B) \vee (Q \wedge \neg B)$ where P_1 is constructed from $\{P_1\}\,C_1\,\{Q\}$.

Logical pretest loops are captured by the Loop Invariant Theorem. The Loop Invariant Theorem uses induction to show that a programmer-supplied *loop invariant* (i.e., a relationship between program variables that remains the same regardless of how many times the loop is executed) holds.

$$\frac{\{P\}\,C\,\{P\}}{\{P\}\,\mathtt{while}(B)\,C; \{Q\}} \qquad \textbf{[Loop Rule]}$$

The selection of an appropriate loop invariant is a difficult task. In section 2.3, we discuss how ProVIDE helps the student construct an appropriate loop invariant by starting with the loop's postcondition and terminating condition and backing over the loop body until a pattern P can be identified. Given an appropriate loop invariant P, the **Loop Rule** tells us that it also serves as the precondition to the loop.

2.2 Constructing Preconditions in ProVIDE

In this section, we illustrate the methodology used by ProVIDE to construct method preconditions. The student begins by entering the program in ProVIDE either from scratch or via the "new class wizard" that prompts the student (via a series of panels) for the class name, the data areas of the class, and the signatures and postconditions for all of the methods of the class. Let's consider the task of squaring an integer N using only addition and subtraction. One possible solution is given below.

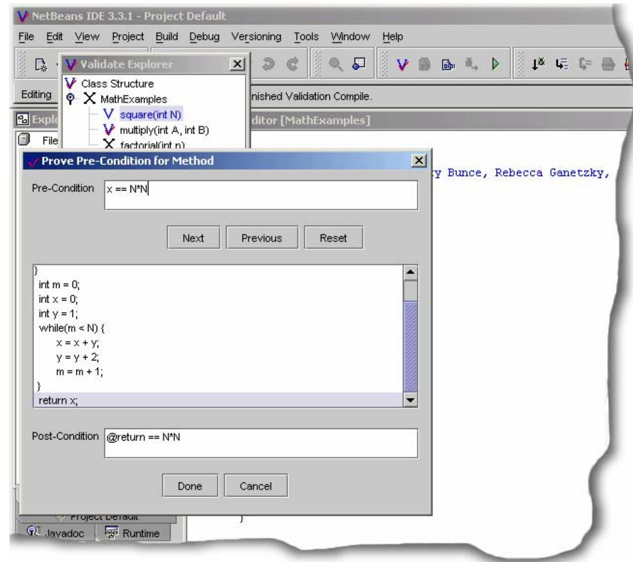
```
/**
 * @author  Tim Gegg-Harrison, Gary Bunce, Rebecca Ganetzky,
 */

public class MathExamples {

    /**
     * @post @return == N*N
     */
    public int square(int N) {
        if (N < 0) {
            N = -N;
        }
        int m = 0;
        int x = 0;
        int y = 1;
        while (m < N) {
            x = x + y;
            y = y + 2;
            m = m + 1;
        }
        return x;
    }
```

Although this method is relatively simple (i.e., it only involves assignment statements, a straightforward conditional statement, and a single bounded loop), it is not at all obvious that it actually computes N^2. For such a method, not only is it important to construct its precondition, it is equally important to provide a convincing argument that it actually works.

Once the student has finished defining the class, he/she presses the "validate" button to obtain ProVIDE's assistance with the construction of preconditions for all of the methods of the class from their student-provided postconditions. Consider the **`square`** method presented above. To construct an appropriate precondition (and prove the correctness of this method), the student begins with the method's postcondition:

$$@return = N \times N$$

which we will denote as $@return = N^2$. ProVIDE helps the student by guiding him/her through a series of dialog windows, the first of which asks the student to find the weakest precondition to the last statement of the method body given this postcondition.



Since **`return x;`** is the last statement of the method, the student is asked to apply the Assignment Axiom (i.e., replace the pseudo-variable *@return* with x) to construct its weakest precondition from the method's postcondition producing:

$$x = N^2$$

The next-to-last statement of the method is a **`while`** loop so ProVIDE helps the student identify an appropriate loop invariant that is both strong enough to imply the postcondition while being weak enough to enable the student to complete the proof of correctness of the method. Given the method's postcondition and the loop's guard, our method will construct the following loop invariant:

$$\left(x = N^2 - (N-m)y - (N-m)(N-m-1)\right) \wedge (N \geq m)$$

For now, let's assume that this is an appropriate loop invariant. We illustrate how ProVIDE helps the student construct this loop invariant in the next subsection. The **Loop Rule** tells us that a loop invariant is an appropriate precondition to **`while`** loop. Thus, the loop invariant becomes the postcondition to the statement immediately preceding the **`while`** loop. Applying the **Assignment Axiom** to the **`int y = 1;`** statement produces:

$$\left(x = N^2 - (N-m) - (N-m)(N-m-1)\right) \wedge (N \geq m)$$

Backing over the **`int x = 0;`** statement produces:

$$\left(0 = N^2 - (N-m) - (N-m)(N-m-1)\right) \wedge (N \geq m)$$

Backing over the **`int m = 0;`** statement produces:

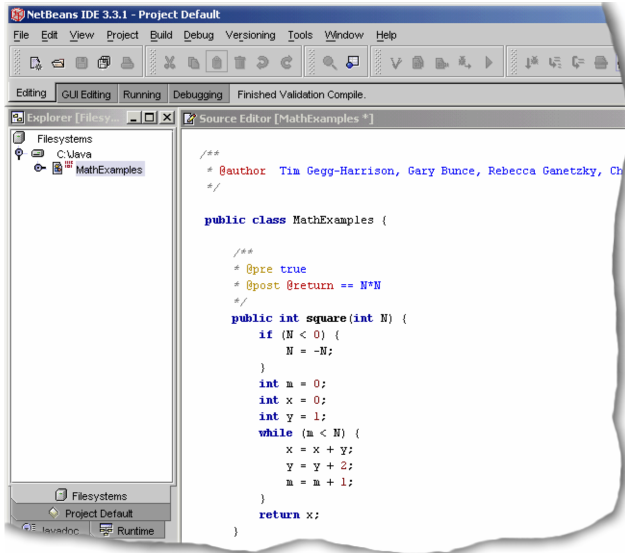
$$\left(0 = N^2 - (N-0) - (N-0)(N-0-1)\right) \wedge (N \geq 0)$$

which can be simplified to $N \geq 0$. Note that if the **`square`** method did not contain the conditional statement then the **`@pre`** tag would become $N \geq 0$. Since it does contain the conditional

statement, however, we must apply the **Conditional Rule** producing:

$$\left(\left(N<0\right)\wedge\left(-N\geq 0\right)\right)\vee\left(\neg\left(N<0\right)\wedge\left(N\geq 0\right)\right)$$

which can be simplified to $\left(N<0\right)\vee\left(N\geq 0\right)$ or simply **`true`**. This states that the **`square`** method is defined for all integers which is appropriate since the purpose of the conditional is to set N to its absolute value. Once the student has completed constructing the precondition, ProVIDE adds the **`@pre true`** Javadoc tag preceding the method definition.



2.3 Constructing Loop Invariants in ProVIDE

ProVIDE guides the student through the construction of an appropriate loop invariant to a **while** loop by starting with the loop's postcondition and terminating condition as the initial postcondition to the loop and backing over each of the statements of the loop body. The loop body is a sequence of statements so ProVIDE recursively opens up a dialog window to guide the student through the construction of an appropriate weakest precondition beginning with the loop's last statement.

After the first backward pass through the loop body, the student has constructed the weakest precondition to the loop body given the loop's postcondition and terminating condition as the initial postcondition to the loop body. This constructed precondition and the loop guard become the postcondition to the second backward pass through the loop body. The weakest precondition that is constructed on the third backward pass through the loop and the loop guard become the postcondition to the fourth backward pass through the loop, etc. After viewing the constructed preconditions for each backward pass through the loop, the student is asked to identify a pattern that is the loop invariant. To ensure that the student's choice of a loop invariant is valid, ProVIDE forces the student through one more backward pass through the loop beginning with the student's proposed loop invariant and asks the student to simplify the constructed precondition to the proposed loop invariant.

For the **`square`** method, **`x == N*N`** is the postcondition of the loop and **`m == N`** is the loop's terminating condition. Thus, we start backing over the loop with the assertion created by their conjunction:

$$\left(x=N^2\right)\wedge\left(m=N\right)$$

Backing over **`m = m + 1;`** we get:

$$\left(x=N^2\right)\wedge\left(m+1=N\right)$$

Since this assertion does not contain the variable **`y`**, backing over **`y = y + 2;`** leaves us with the previous assertion:

$$\left(x=N^2\right)\wedge\left(m+1=N\right)$$

Backing over **`x = x + y;`** we get:

$$\left(x+y=N^2\right)\wedge\left(m+1=N\right)$$

In order to prepare for the second reverse pass, we back over the loop guard. Backing over the guard of a **`while`** loop is like backing over the guard of an **`if-else`** statement resulting in the conjunct formed from the guard and the assertion to this point:

$$\left(x+y=N^2\right)\wedge\left(m+1=N\right)\wedge\left(m<N\right)$$

But this simplifies back to the previous assertion:

$$\left(x+y=N^2\right)\wedge\left(m+1=N\right)$$

Backing over **`m = m + 1;`** we get:

$$\left(x+y=N^2\right)\wedge\left(m+2=N\right)$$

Backing over **`y = y + 2;`** we get:

$$\left(x+y+2=N^2\right)\wedge\left(m+2=N\right)$$

Backing over **`x = x + y;`** we get:

$$\left(x+y+y+2=N^2\right)\wedge\left(m+2=N\right)$$

This can be simplified to:

$$\left(x+2y+2=N^2\right)\wedge\left(m+2=N\right)$$

In order to prepare for the third reverse pass, we add the loop guard to this assertion when we back over it:

$$\left(x+2y+2=N^2\right)\wedge\left(m+2=N\right)\wedge\left(m<N\right)$$

But this simplifies back to the previous assertion:

$$\left(x+2y+2=N^2\right)\wedge\left(m+2=N\right)$$

On the third reverse pass, backing over **`m = m + 1;`** we get:

$$\left(x+2y+2=N^2\right)\wedge\left(m+3=N\right)$$

Backing over **`y = y + 2;`** we get:

$$\left(x+2\left(y+2\right)+2=N^2\right)\wedge\left(m+3=N\right)$$

Backing over **`x = x + y;`** we get:

$$\left(x+y+2\left(y+2\right)+2=N^2\right)\wedge\left(m+3=N\right)$$

This can be simplified to:

$$\left(x+3y+6=N^2\right)\wedge\left(m+3=N\right)$$

It is easy to show that a generalized loop invariant for the kth backward pass is:

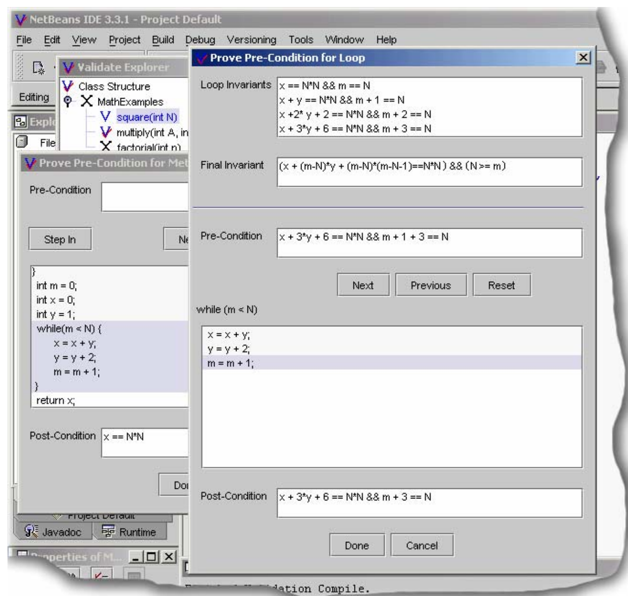
$$\left(x+ky+k\left(k-1\right)=N^2\right)\wedge\left(m+k=N\right)$$

Since $m+k=N$, we know that $k=m-N$, so we can rewrite this generalized assertion as:

$$\left(x+\left(m-N\right)y+\left(m-N\right)\left(m-N-1\right)=N^2\right)\wedge\left(k=m-N\right)$$

The terms including *k* no longer contribute to this assertion so we remove them, giving us the loop invariant:

$$\left(x+\left(m-N\right)y+\left(m-N\right)\left(m-N-1\right)=N^2\right)\wedge\left(N\geq m\right)$$



3 Conclusion

Mathematics is an integral part of science, mathematics, engineering, and technological (SMET) education. In particular, mathematics is fundamental to computer science. Our experience has shown an improvement in student learning as a result of an active attempt to make mathematics more relevant. Preliminary results indicate that ProVIDE will further enhance student learning by providing the seamless integration of analysis with the creation of computer programs. By incorporating mathematical analysis tools within ProVIDE, students are able to practice "learning by doing", applying an apprenticeship learning [3], and learning in a "situated cognition" environment [1,2]. Although we will not be conducting a formal assessment until this fall, our preliminary assessment of the impact of ProVIDE in the classroom is promising.

Acknowledgments

This research was supported in part by NSF DUE-0127483.

References

[1] Brown, J.S., Collins, A., and Duguid, P. Situated Cognition and the Culture of Learning. *Educational Researcher*, 18(1): 32-42, 1989.

[2] Cognition and Technology Group at Vanderbilt. Anchored Instruction and its Relationship to Situated Cognition. *Educational Researcher*, 19(6): 2-10, 1990.

[3] Collins, A., Brown, J.S., and Newman, S.E. Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics. In L.B. Resnick, editor, *Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser*, pages 453-494, Hillsdale, NJ: Lawrence Erlbaum, 1989.

[4] Dijkstra, E.W. *A Discipline of Programming*. Upper Saddle River, NJ: Prentice Hall, 1976.

[5] Floyd, R.W. Assigning Meaning to Programs. In J.T. Schwartz, editor, *Proceedings of the Symposium on Applied Mathematics (Mathematical Aspects of Computer Science)*, American Mathematical Society, Providence, Rhode Island, 1967.

[6] Gegg-Harrison, T.S. Ancient Egyptian Numbers: A CS-Complete Example. In J. Gersting and R. McCauley, editors, *Proceedings of the 32nd Technical Symposium on Computer Science Education*, pages 268-272, Charlotte, North Carolina, February 2001 (*ACM SIGCSE Bulletin*, 33(1): 268-272, 2001).

[7] Gegg-Harrison, T.S., Bunce, G.R., Ganetzky, R.D., Olson, C.M., and Wilson, J.D. Studying Program Correctness in ProVIDE. Software demonstration at the *8th Annual Conference on Innovation and Technology in Computer Science Education*, Thessaloniki, Greece, June–July 2003.

[8] Gries, D. *The Science of Programming*. New York, NY: Springer−Verlag, 1981.

[9] Hoare, C.A.R. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10): 576-580, 1969.

[10] Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science*, Final Report, December 2001.

[11] Kramer, R. iContract – The Java Design by Contract Tool. In M. Singh, B. Meyer, J. Gil, and R. Mitchell, editors, *Proceedings of the 26th Conference on Technology of Object-Oriented Languages and Systems*, pages 295-307, Santa Barbara, California, August 1998.

[12] Meyer, B. *Eiffel: The Language*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[13] Meyer, B. Applying "Design by Contract". *IEEE Computer*, 25(10): 40-51, 1992.