SCREEN-REPLAY: A TOOL FOR TRACKING HOW STUDENTS DEVELOP PROGRAMS WITH HTDP

by

Mehmet Fatih Köksal

BS, Computer Science, Istanbul Bilgi University, 2006

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2010

ACKNOWLEDGEMENTS

I would like to thank all people who have helped and inspired me during my master study.

I especially want to thank my advisor, Dr. Suzan Üsküdarlı, for her guidance during my research. This thesis would not have been possible without the help, support and patience of her.

My deepest gratitude goes to my wife and my family for their unflagging love and support throughout my life. I am extremely grateful for their patience.

It is a pleasure to thank Prof. Matthias Felleisen, other members of PLT group and Chris Stephenson, head of Istanbul Bilgi University Computer Science Department, who made this thesis possible.

I would like to thank Remzi Emre Başar, Vehbi Sinan Tunalıoğlu, Bülent Özel and other members of Computer Science Department of Istanbul Bilgi University. They have made their support available in a number of ways.

Finally, I want to thank my students and all other kind people around me for their help and support during my research.

ABSTRACT

SCREEN-REPLAY: A TOOL FOR TRACKING HOW STUDENTS DEVELOP PROGRAMS WITH HTDP

Evaluation of the teaching method has great importance in improving the course quality. This evaluation is harder in courses which focus on the process of program development, since it requires observation of the students' approach to problem solving. HtDP offers a "design recipe" which focuses on the process of program development. While there have been a number of studies focusing on the quality of this approach, there has not been any quantitative analysis.

In this study, I first introduce a model and implementation of a tool (Screen-Replay) that enables the recording, replaying and annotation of programming sessions. This tool is implemented for DrScheme environment using Scheme programming language. It records and replays a programming session exactly as it occurred. Furthermore, while replaying, an observer may annotate the programming session by associating HtDP design recipe steps with specific time intervals. The resulting annotations form a sequence of design activity descriptions which describe the development process. In order to assess these sequences, a process scoring algorithms is proposed. Finally, the process scores and exam grades from a set of 61 development sessions are examined to gain insight into the impact of following design recipe on exam grades.

Screen-Replay was effective for observing how students develop their programs. In contrast to personal observation, this approach provided consistent and objective observation of students development processes.

ÖZET

SCREEN-REPLAY: HTDP KULLANAN ÖĞRENCİLERİN PROGRAM GELİŞTİRME SÜREÇLERİNİ TAKİP ETME ARACI

Öğretim metodlarının değerlendirilmesi ders kalitesinin arttırılması bakımından büyük önem taşır. Program geliştirme süreçlerine odaklanmış dersler öğrencilerin problem çözme yöntemlerinin gözlemlenmesini gerektirdiği için öğrenim metodlarının değerlendirilmesi bu derslerde daha zordur. HtDP de bu tarz bir sürece odaklanan "tasarım reçetesi" önermektedir. Bu yaklaşımın kalitesini ölçmek için yapılmış çalışmalar olmasına rağmen nicel analizine ilişkin bir çalışma henüz yapılmamıştır.

Bu çalışmada ben öncelikle program geliştirme sürecinin kaydedilmesi, tekrar oynatılması ve üzerine notlar alınmasını sağlayan bir araç (Screen-Replay) modeli ve uygulaması ortaya koymaktayım. Bu araç DrScheme ortamı için Scheme programlama dili kullanılarak yazılmıştır. Program geliştirme sürecini kaydederek aynen tekrar oynatılabilmesini sağmaktadır. Ayrıca, süreci tekrar oynatan bir gözlemci belirli zaman aralıklarını HtDP tasarım reçetesi adımları ile eşleştirmek sureti ile süreci yorumlayabilir. Sonuçta ortaya çıkan yorumlar, geliştirme sürecini tanımlayan tasarım aktivitelerinin dizisidir. Bu dizileri değerlendirebilmek için bir süreç değerlendirme algoritması geliştirildi. Son olarak, 61 farklı program geliştirme sürecinden elde edilen süreç skorları ve sınav notları incelenerek tasarım reçetelerinin sınav notları üzerindeki etkisi kavramaya çalışıldı.

Screen-Replay, öğrencilerin nasıl program geliştirdiğini gözlemlemek için etkili bir araçtır. Kişisel gözlemleme yöntemlerine karşın öğrencilerin geliştirme süreçlerinin tutarlı ve nesnel bir yöntemle gözlemlenmesini sağlamıştır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . viii
LIST OF TABLES . . . x
LIST OF ABBREVIATIONS . . . xii
1. INTRODUCTION . . . 1
2. BACKGROUND . . . 4
2.1. PLT and the TeachScheme! Project . . . 4
2.2. Language Levels . . . 5
2.3. DrScheme . . . 6
2.4. The Design Recipe and HtDP . . . 8
3. MOTIVATION . . . 12
3.1. From Product-Oriented to Process-Oriented Approach . . . 12
3.2. Current First Year Curriculum . . . 16
3.3. The Strategic War Between Instructors and Students . . . 17
3.4. The Need for the Observation of the Process . . . 18
3.5. Observation without Personal Intrusion . . . 19
4. IMPLEMENTATION OF SCREEN-REPLAY . . . 21
4.1. The Basic Approach . . . 21
4.2. Requirements . . . 23
4.3. Implementation . . . 24
4.3.1. Recorder . . . 25
4.3.2. Replayer . . . 26
4.3.2.1. Replaying . . . 28
4.3.2.2. Tagging . . . 30
4.4. Processing the Tags . . . 33
5. PROCESS SCORING ALGORITHM . . . 36
5.1. Assessing the Design . . . 36

5.2. Classical String Similarity Algorithms 37
5.3. Our Scoring Algorithm 39
6. EXPERIMENT 43
6.1. Experimental Questions 43
6.2. Participants 43
6.3. Tasks 44
6.4. Monitoring 45
6.5. Grading the Exam 45
7. EVALUATION 50
7.1. Question 1 50
7.2. Question 2 51
7.3. Question 3 55
8. DISCUSSION 58
8.1. Screen-Replay 58
8.2. Process Scoring Algorithm 59
8.3. Experiment 59
9. RELATED WORK 61
10. FUTURE WORK 64
11. CONCLUSION 65
APPENDIX A: Experiment Data: Process Scores and Exam Grades 66
REFERENCES 69

LIST OF FIGURES

Figure 2.1. DrScheme's interface 7

Figure 2.2. Basic steps of the design recipe 8

Figure 2.3. An example application of design recipe 10

Figure 3.1. The RobotWorld applet 13

Figure 3.2. Example code for JAVA + RobotWorld 14

Figure 4.1. A programming session tagged with design step tags. The entire session starts at t_0 and terminates with t_n and sub-processes occur in sub-intervals. 21

Figure 4.2. Overview of tracking and assessing programming session 22

Figure 4.3. The definition and example of the action structure 25

Figure 4.4. Start recording (a) and stop recording (b) buttons 27

Figure 4.5. An overview of the Replayer 27

Figure 4.6. The definition of the tape structure 29

Figure 4.7. The definition and example of tag structure 31

Figure 4.8. An example output of the tagging process 32

Figure 4.9. The definition of the processed-tag structure 34

Figure 5.1. Pseudo-code of the process scoring algorithm 41

Figure 6.1. Exam questions for session 1 and session 2 44

LIST OF TABLES

Table 3.1. Available built-in functions for the RobotWorld program 15

Table 4.1. Design steps and associated tags 30

Table 4.2. Students actions and observers responses in return 35

Table 5.1. Evaluation of the process scoring algorithm 42

Table 6.1. Demographic information of exam participants 43

Table 6.2. Grading each function using the scala 46

Table 6.3. Scale grades converted to number grades over 100 47

Table 6.4. Percentages of design step weights for main and helper functions . 48

Table 6.5. Application of design step weights 48

Table 6.6. Calculation of the final grade 48

Table 7.1. Categorization of the exam grades 52

Table 7.2. Process score categorization for the main function 52

Table 7.3. Process score categorization for helper functions 52

Table 7.4. Process scores vs. exam grades for main function 53

Table 7.5. Process scores vs. exam grades for helper functions 53

Table 7.6. Process scores vs. exam grades for helper functions (excluding zero values) 54

Table 7.7. Process scores vs. exam code grades for main function 54

Table 7.8. Process scores vs. exam code grades for helper functions (excluding 0's) 54

Table 7.9. Exam grade vs. design step category 55

Table 7.10. Frequently used transactions by all students (over 5 per cent) . . . 57

Table 7.11. Frequently used transactions by successfull students (over 5 per cent) 57

Table A.1. Process scores and exam grades of students (1) 66

Table A.2. Process scores and exam grades of students (2) 67

Table A.3. Process scores and exam grades of students (3) 68

LIST OF ABBREVIATIONS

CD	Code
COMP111	Introduction to Programming 1
CS	Computer Science
CT	Contract
EX	Examples
HtDP	How to Design Programs
IDE	Integrated Development Environment
LGPL	GNU Lesser General Public License
PLT	A research group founded by Prof. Matthias Felleisen (not an abbreviation)
PLT-Scheme	A programming language based on Scheme
PP	Purpose
rec	Source file that includes recorded process data
scm	Scheme language source code file
TL	Template
TS	Test

1. INTRODUCTION

The education of a computer science student usually starts with an introductory programming course. The aim of such courses is to equip students with general programming knowledge and prepare them for subsequent courses in the curriculum. Such courses typically teach the fundamental concepts of programming with the use of given programming language, integrated development environment (IDE), and other tools [1]. With these tools and course instructions, students are expected to learn how to write, debug and document programs.

While objectives of introductory programming courses are similar, contents, approaches and assessment methods differ. Teaching with examples is a frequently used approach [2], where examples are provided for every concept introduced. These examples are expected to guide students in their assignments. Students often use these examples as a starting point and modify them until they reach the desired solution. Conventional assessment methods evaluate exams and assignments by comparing students' code against expected results. Students' code, in such cases, are final products with no further information on how they arrived to them.

The TeachScheme! project [3] does not support the programming-by-tinkering approach, which students often resort to. It developed an alternative approach to teaching, described in the text book "How to Design Programs" (HtDP) [4]. This approach focuses on a design process that starts from the problem statement and proceeds to a well-organized solution. After the publication of HtDP, several universities around the world revised their curriculum in favor of this approach. Most of these universities use the methodology as described in the book, whereas some [1] have derived alternate versions [5] to meet their needs.

The HtDP approach and others derived from it emphasize the importance of *process* in addition to the product. Accordingly, instead of conventional assessment methods, they prefer lab (or live) exams, which they consider to be a more accurate

reflection of student performance [6]. Approaches to conduct live exams also vary. Some let students develop programs independently and evaluate the results in a conventional manner. In others [1, 7], the development process is personally observed.

In order to understand how students develop their programs, it is necessary to track their development process. By tracking their process, we aim to answer the following questions: Do students apply the design guidelines we teach, when they develop programs on their own? Are students, who follow the suggested guidelines, more successful than the others? If not, can we identify patterns or approaches used by successful students?

Intrusive tracking, such as watching, may impact students' performance during a programming session. Indeed, it has been reported that some students were disturbed by personal observation of their work [7]. An alternative approach for observing program development is to embed tracking into the development tool. Such a tool would need to record and replay the development process in order to make the process transparently visible.

In this study, I first introduce a model and implementation of a tool (ScreenReplay) that enables the recording, replaying and annotation of programming sessions. This tool is implemented for DrScheme [8] environment using Scheme [9] programming language. It records and replays a programming session exactly as it occurred. Furthermore, while replaying, an observer may annotate the programming session by associating HtDP design recipe steps with specific time intervals. The resulting annotations form a sequence of design activity descriptions which describe the development process. In order to assess these sequences, a process scoring algorithms is proposed. Finally, the process scores and exam grades from a set of 61 development sessions are examined to gain insight into the impact of following design recipe on exam grades.

The rest of this thesis is organized as follows: Chapter 2 presents some background information on the design recipe and HtDP. Chapter 3 further discusses my motivation to analyze students' programming sessions. Chapter 4 describes the model and

implementation details of the Screen-Replay. Chapter 5 explains the process scoring algorithm. Chapter 6 presents the details of the experiment, followed by an evaluation in Chapter 7. Chapter 8 and Chapter 9 include discussions and related work, respectively. Finally, in Chapters 10 and 11, I present future work to be done and conclude my work.

2. BACKGROUND

This chapter presents background information needed to follow this work. Primarily, the objectives and approach of the TeachScheme! [3] project are explained.

2.1. PLT and the TeachScheme! Project

PLT was founded by Prof. Matthias Felleisen [10] in the mid 1990s, as a research group at Rice University. The aim was to produce pedagogic materials for novice programmers. In 1996, PLT started the TeachScheme! project [3]. The objective of this project was to reform introductory high school courses on programming with the long-term goal of overcoming all the problems they have diagnosed. Three major problems presented by the authors [3] are as follows:

(i) Schools consider programming as a part of vocational studies rather than the liberal arts core. Therefore, in many places "introduction to computer science" is a course on application software and has nothing to do with programming and computing.

(ii) Schools employ programming technology from the industry that is just the cheapened version of the professional products, hence, is not appropriate for educational purposes.

(iii) The curriculum of high schools often dictate the grammar of currently fashionable, vocational programming languages rather than teaching principles of design and problem solving.

The following three sections present major contributions of TeachScheme! project concerning these problems stated above.

2.2. Language Levels

Beginners have a lot of errors (of all types) in their programs and easily get frustrated, when the feedback for errors is obscure [3]. Therefore, it is critical for an introductory course on programming to present ideas and use tools that help students to comprehend and overcome these errors.

After comprehensive observation of lab sessions at Rice University and in local high schools in Houston, the TeachScheme! project designed a series of programming languages based on Scheme [3][1] . Each element of the series corresponds to a cognitive stage of the learning process in a concrete manner:

- Beginning Student: This is a small version of Scheme that is tailored for beginning computer science students.
- Beginning Student with List Abbreviations: This language is an extension to "Beginning Student" that prints lists with *list* instead of *cons*, and accepts *quasiquote*d input.
- Intermediate Student: The Intermediate Student language adds local bindings and higher-order functions.
- Intermediate Student with Lambda: This language adds anonymous functions.
- Advanced Student: The Advanced Student language adds mutable state.

When programs in these languages are run in DrScheme, any part of the program that was not run is highlighted in orange and black, indicating that those parts of the program have not been tested. To avoid these colors, students are expected to write test cases. Seeing no colors does not mean that the program is fully tested, but it simply means that each part of the program has been run (at least once) [11].

Providing a coherent series of sub-languages instead of a single language has two major advantages: First, by starting with a small-language, teachers can focus on the

[1]Original paper introduces three language levels, but, due to additional observations TeachScheme! project currently works with five teaching languages.

development of problem solving, instead of discussing the appropriate linguistic mechanisms (such as using "for" as well as "while"). Second, using a specific sub-language enables the language implementors to report error messages that are appropriate to a student's level of knowledge.

2.3. DrScheme

The classroom observations of TeachScheme! team also revealed problems with the IDE's used for the introductory programming courses. Complex control panels are one of the main problems that confuse beginners [3]. Learning editions of these IDE's also do not help, since they are often cheapened editions of professional tools which include the same complex control panels.

TeachScheme! project developed DrScheme as a response to these issues. DrScheme has a very pure interface with two windows for definitions and interactions, along with 5 carefully chosen buttons (see Figure 2.1). It supports syntax highlighting, parenthesis matching and other functionalities to help novice users. It includes a debugger, macro stepper [12] and a syntax checker. The stepper helps a teacher to explain easily a complete model of computation to students without ever mentioning a hardware concept.

Another important feature of DrScheme is that it supports language levels explained in Section 2.2. Choosing the appropriate language level allows students to benefit from more explanatory error messages.

DrScheme is published under the GNU Lesser General Public License (LGPL) [13]. Thus, it can be modified, redistributed or linked into any other application as long as the rules of LGPL are followed. This makes DrScheme freely available and allows instructors to modify it according to their needs.

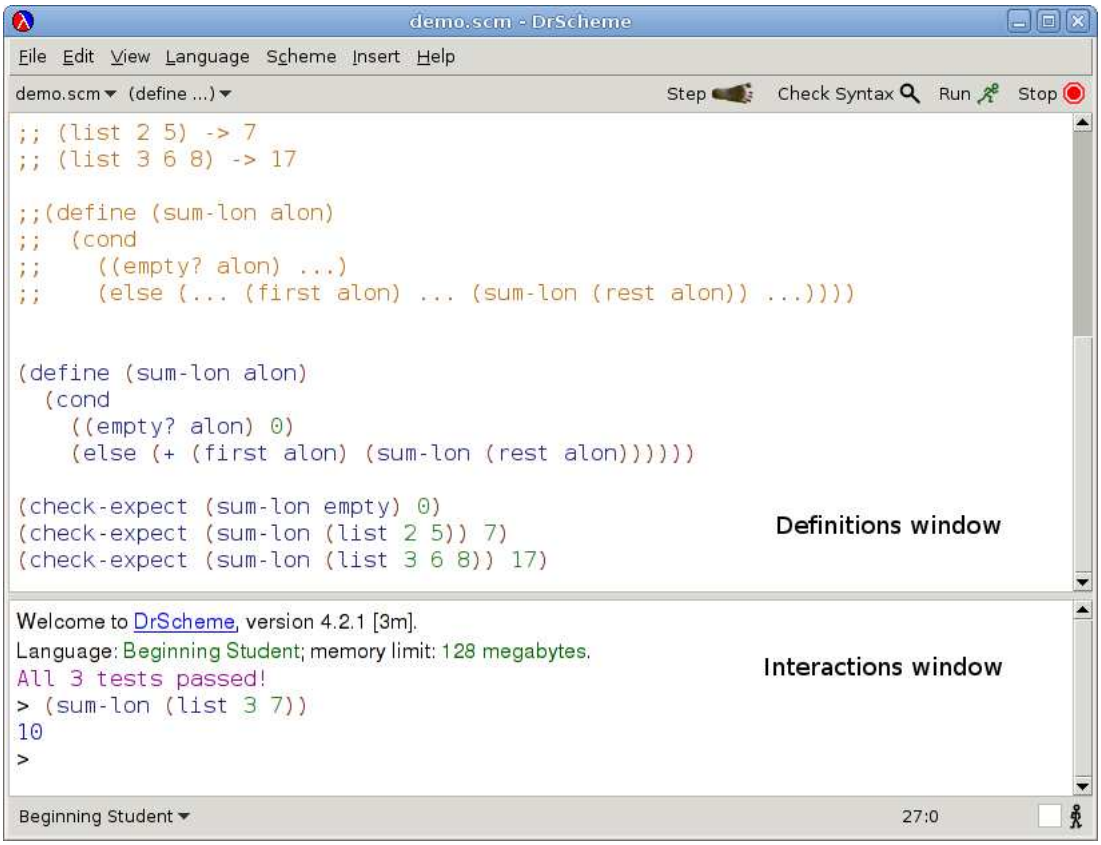

Figure 2.1. DrScheme's interface

2.4. The Design Recipe and HtDP

TeachScheme! project emphasizes the importance of process in addition to the product. It introduces a program design method to help beginning students and their teachers. This design method is extensively described in the book HtDP as a teaching methodology.

HtDP is defined by its authors as "... the first book on programming as the core subject of a liberal arts education". It focuses on the design process that leads from problem statements to well-organized solutions rather than studying the details of a specific programming language, algorithmic minutiae, and specific application domains [4]. It includes design guidelines, which are formulated as a number of *program design recipes* leading students from a problem statement to a computational solution in a step-by-step fashion with well-defined *intermediate products*.

A design recipe is a checklist that helps students to organize their thoughts through the problem solving process. Students are expected to use this checklist on a question-and-answer basis to progress towards a solution [4]. The basic steps of the design recipe are shown in Figure 2.2.

0. *(DD) Data definition*: determine the classes of data that the program consumes and produces
1. *(CT) Contract*: name your function and specify the input-output relation in terms of the defined data type
2. *(PP) Purpose*: informally specify *what* the program is to compute
3. *(EX) Examples*: illustrate the behavior with examples
4. *(TL) Template*: describe your programs template/layout
5. *(CD) Code*: transform your template into a complete definition
6. *(TS) Tests*: turn your examples into formal test cases

Figure 2.2. Basic steps of the design recipe

The version of design recipe presented here includes 7 steps, where the original one has 6. In this version[2] , purpose statement and the contract are split into two different steps. It starts from 0, since the data definition may be used by a number of different functions, while the other steps are function specific.

It is the best to explain the design recipe with an example. The example in Figure 2.3 illustrates how to apply the design recipe to the problem of summing elements of a list.

One may ask, whether it is necessary to write each step down during the development process. A student, still, may write a well structured, working program, without including all intermediate steps, but having them in mind as he/she proceeds. Writing each step separately has a lot of advantages for (1) the student, (2) his/her teacher and (3) those who use the students program at some future time.

First of all, it helps students to overcome the so called "blank page syndrome" [14, 15]. Given a problem statement and a white blank paper, students do not have to think about where to start. They can immediately start by asking themselves the following question: "Which classes of data does this program consume and produce?", that leads to a data definition. This question will then be followed by other design recipe questions and guide the student towards a well-structured solution.

Second, it helps teachers. Every design step the student produces is a verifiable intermediate product. When a student comes to the teacher, because he got stuck, the teacher can ask questions about the intermediate design steps and try to understand the real problem that the student is facing. Thus, the teacher is able to lead students in the right direction, instead of trying to correct an erroneous code piece. Similarly, a teacher can use these intermediate products for grading and evaluation. Instead of guessing how many points to assign to a correct statement in a program, the teacher can inspect the process that produced the product and assign a more accurate grade [3].

[2]as it is used in Istanbul Bilgi University Department of Computer Science

```
;; Data definition:                                                    0
;; a list of numbers (lon) is either;
;; 1. empty, or
;; 2. a pair of
;;    a) a number and
;;    b) a list of numbers (lon)

;; Contract:                                                           1
;; sum-lon: lon -> number

;; Purpose:                                                            2
;; this function consumes a list of numbers and produces
;; the sum of the elements of the given list

;; Examples:                                                           3
;; empty         -> 0
;; (list 5)      -> 5
;; (list 3 1)    -> 4
;; (list 4 7 -2) -> 9

;; Template:                                                           4
;; (define (sum-lon alon)
;;   (cond
;;     ((empty? alon) ...)
;;     (else
;;       ... (first alon) ... (sum-lon (rest alon)) ...)))

;; Code:                                                               5
(define (sum-lon alon)
  (cond
    ((empty? alon) 0)
    (else (+ (first alon) (sum-lon (rest alon))))))

;; Tests:                                                              6
(check-expect (sum-lon empty) 0)
(check-expect (sum-lon (list 5)) 5)
(check-expect (sum-lon (list 3 1)) 4)
(check-expect (sum-lon (list 4 7 -2)) 9)
```

Figure 2.3. An example application of design recipe

Third, it helps other people who will later use the students code. Design recipe A program that is produced using the design recipe contains all the necessary documentation, and is tested.

Finally, it is a good practice to develop solutions of any kind of problems. Students internalize the approach and become able to use this methodology to solve problems in other areas or to build more complex systems.

The design recipe approach has been gradually adopted and used by the Computer Science Department of Istanbul Bilgi University [16, 17]. The following chapter explains this adaptation process, difficulties encountered and issues that motivated this study.

3. MOTIVATION

3.1. From Product-Oriented to Process-Oriented Approach

The Computer Science (CS) Department of Istanbul Bilgi University was founded in 1999. Since its foundation, the introductory programming curriculum of the department has changed three times[3] .

Starting from 1999/Fall semester, the CS department used C programming language for two years in the introductory programming course (COMP111). This course used a conventional C language curriculum. For every concept introduced, students were provided with example solutions. These examples were expected to guide students in their assignments. Students typically used these examples as starting points and modified them (in a code and fix manner) until they reached the desired solution. Instructors evaluated assignments and exams by conventional methods, where students code was compared against the expected result. It was observed that students were mostly struggling with the heavy syntax of the language instead of learning how to solve problems. Therefore, the CS department ceased using the C programming language in COMP111.

The 2001/Fall semester started with the use of a different programming language, JAVA. Instead of teaching "how to write programs in JAVA", COMP111 aimed to teach "how to solve problems". Introductory programming concepts were introduced to students by using a program, called RobotWorld [18]. The RobotWorld program was developed by Chris Stephenson (head of the CS department). The aim was to relieve students from the unnecessary syntactic issues of the programming language, and to allow them to concentrate on problem solving skills (like in [19, 20, 21, 22, 23]). Basically, the program dealt with a checker board and a blind robot. Students had to instruct the robot in accomplish some tasks, which vary from finding the center of the

[3]The author has been a student and teaching assistant at Computer Science Department of Istanbul Bilgi University

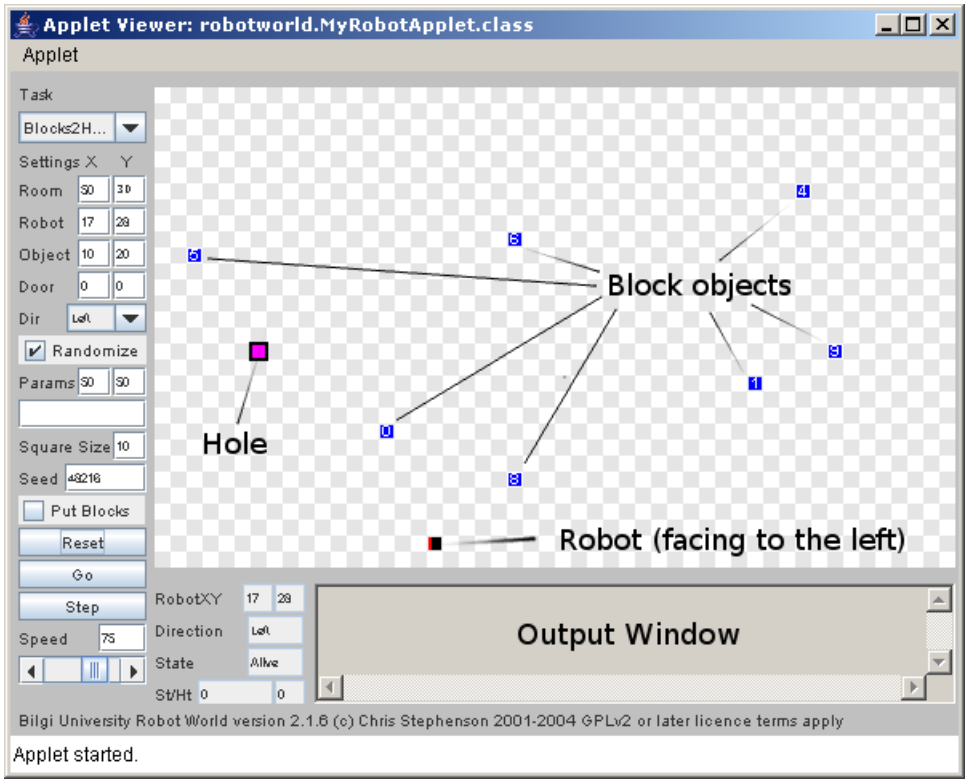

Figure 3.1. The RobotWorld applet

room to putting block objects in to the hole (see Figure 3.1 and 3.2). The robot was not only blind, but it understood a small number of instructions (see Table 3.1). Accomplishing complex tasks were only possible by building new functions upon already existing ones. For example, to make the robot turn 90° to the right, students turned it 3 times 90° to the left. This new functionality then, was saved for future use. In this way, student learned the concept of re-use through experience rather than a formal introduction of the concept.

Instructors of the CS department considered JAVA + RobotWorld to be more appropriate than the C programming language for COMP111. Firstly, it reduced the syntactic overhead and lowered the learning curve. Secondly, it was more successful than C in learning the introductory programming concepts. On the other hand, JAVA + RobotWorld had some issues as well. Some students were, still, struggling with syntactic issues for the initial 3-4 weeks of the term. They used the language constructs without understanding what they mean (like static, public, void etc.). The main effects in Robotworld were obtained through *side effects*, that is, a move of the robot was

```
import robotworld01.*;
class MyRobotProgram extends RobotProgram
{
   void walkToTheWall()  throws RobotResetException
   {
      if (tryStep())
         walkToTheWall();
   }
   int countToTheWall()  throws RobotResetException
   {
      if (tryStep())
         return 1 + countToTheWall();
      else
         return 0;
   }
   int findTheWidth() throws RobotResetException
   {
      walkToTheWall();
      doTurn();
      doTurn();
      return 1 + countToTheWall();
   }
   public void go(String task, int x, int y,String s)
   throws RobotResetException
   {
      print (``I took ``+countToTheWall()+'' steps'');
      print(``the room is ``+findTheWidth()+'' steps wide'');
   }
}
```

Figure 3.2. Example code for JAVA + RobotWorld

Table 3.1. Available built-in functions for the RobotWorld program

print(String)	prints any String
tryStep()	tries to step the robot one step forward; the return value is true if the step is successful, false if the Robot hits an obstacle and cannot move forward
doTurn()	turns the Robot 90 degrees to the left, which is always possible
setPainting(boolean)	sets the robot to paint if true, not to paint if false
getPainting()	gets the current painting setting
getObjectNumber()	looks for an object in the current square, gives its number if there is an object, otherwise -1
getBlockNumber()	looks for a block in front of the Robot, gives its number if there is a block in front of the Robot, otherwise -1
void hideRobot()	hide the robot
void showRobot()	show the robot
void hideCheckerBoard()	hide the checker board effect in RobotWorld
void showCheckerBoard()	show the checker board effect in RobotWorld

not a value of a function, but simply the side effect of the evaluation of the function[4] . Therefore, proper design and proper automated testing were impossible. Moreover, the weekly assignments of RobotWorld were far from being real world applications. Students did not feel ready for an internship or the application for any other programming language at the end of the year. They were not able to transfer their programming knowledge into other problem domains, and it was obvious that companies were not using robots. Finally, when students got stuck, helping them was not easy for instructors. They had to inspect the students code and find the bug, without having much idea about the students understanding of the problem.

After 4 years of JAVA + RobotWorld, COMP111 switched to PLT-Scheme [24], a programming language based on Scheme [9], and gradually adopted the HtDP cur-

[4]This information is obtained from Chris Stephenson through e-mail conversations.

riculum. Adaptation of the new curriculum took 2 years. Following this change, the course structure of the department also revised. The next section explains the current introductory programming curriculum in more detail.

3.2. Current First Year Curriculum

The first year curriculum of Computer Science Department at İstanbul Bilgi University was revised effective of 2007-2008 academic year. In order to have better control over the course and increase student-instructor interaction, sections of at most 20 students were opened. With this change, it became possible to intensively follow students to see if they meet the educational objectives.

The new introductory programming course (Comp149/150-HtDP) at İstanbul Bilgi University, is a part of the meta-course Comp149/150, which also includes the courses: Academic Skills (Comp149/150-AS), Meta Skills (Comp149/150-MS) and Discrete Mathematics (Comp149/150-DM). This meta-course is mandatory for Computer Science, Financial Mathematics and Business Informatics majors. Comp149/150-HtDP uses "How to Design Programs" (HtDP) [4] as the text book, Scheme as the programming language and DrScheme [8] as the development environment.

The first semester of the course (Comp149-HtDP) covers first four parts of the book, which basically includes primitive, compound and recursive data types, conditionals, and abstraction. Generative recursion, graphs, vectors and iterative programming are taught in the second semester (Comp150-HtDP).

Each semester consists of 13 weeks. Every week there are two hours of lectures and two hours of labs. In lecture hours, instructors present the material and develop programs in front of the students by following the design recipe as suggested by HtDP. Additionally, each week students are assigned a project, which they must complete within one week. In the final weeks of the second semester, assignments become more complex and students are given a minimum of two weeks to complete them. During lab sessions students present their project solutions to their classmates.

During this course students are given four live exams, where they develop programs in computer labs. Each exam consists of one or two questions that are to be solved in approximately 90 minutes. During the exam, students have access only to the text book and DrScheme. Network access is disabled during exams. Grades of weekly projects and live exams determine the course grade of students. The final grade this course is combined with grades from other parts of the meta-course with a formula that rewards even performance. This grading policy was established based on the belief that students must have sufficient knowledge of mathematics, critical reading/thinking skills and the ability to express their thoughts properly in order to develop well structured programs. Starting from the 2008-2009 academic year, at the end of the year students are examined by a jury consisting of the meta-course instructors.

The main objective of the overall course is to teach "How to solve it?" [25] and the process is the central concern in this idea. The HtDP curriculum meets the aim of the introductory programming course, as it focuses on problem solving process via design recipes, rather than the product. Additionally, it offers a more appropriate support to the introductory curriculum, by using DrScheme and different language levels. However, it introduces some challenges with respect to evaluation, which are discussed in the following section.

3.3. The Strategic War Between Instructors and Students

There are numerous reports on the success using the HtDP curriculum [26, 1, 27, 28]. Since the adoption of HtDP, CS department of Istanbul Bilgi University also observed similar improvements. Specifically, improvements in student performance have been observed with respect to:

- programming abilities,
- overall grades, and
- success in subsequent courses.

These improvements are particularly noticeable in female students [2].

On the other hand, increased interaction with students revealed some issues with respect to the adoption of the taught process. It became apparent that students were not applying the design recipe throughout the development process. Instead, they were diving into the code without going through each step of the design recipe. So, a change in the grading policy was implemented. The grading scheme was modified to grade each step of the design recipe separately.

Students responded to the new grading scheme by faking the process. They wrote the code first and inserted the design steps later. This response led us to inspect each student submission more carefully. Forged design steps are often distinguished by checking the inconsistencies between written design steps. Considering that, a consistency check between design steps and terminating the evaluation of the assignment when an inconsistency was found, was implemented. Unfortunately, this also did not resulted in following the design recipe.

At that point, it became apparent that evaluating the final product and making assumptions about the development process was not the proper way of evaluating a process oriented approach. Applying more pressure to following the design recipes only created better "design recipe evasion" tactics.

With this realization the attempt to evaluate the order of construction by looking only to the final product was abandoned. However, the interest in tracking our student development process and its impact on successful outcome remained strong. The next section further discusses the need for the observation of the development process.

3.4. The Need for the Observation of the Process

The evaluation method of a course should be consistent with the course objectives. There are two main reasons for that: First of all, it ensures that we are evaluating what we are teaching. Second, it helps improving teaching methods. Therefore, a course that aims to teach the process of program development should evaluate students program development.

HtDP, as a process oriented approach for the introductory programming curriculum, should observe students program development processes. Students may, as have been experienced, write the program code and then add other design steps to get higher grades. Such students, obviously, do not benefit from the design recipe as aimed by course objectives. Also, sufficient feedback to improve teaching method is not available. Final products are not sufficient for observing design steps that students have difficulties in applying.

In conclusion, final product evaluation is insufficient in evaluating process oriented approaches. Instead, the development processes must be observed in order to better evaluate. The next section discusses the problems of current observation methods and proposes a different one instead.

3.5. Observation without Personal Intrusion

There are several ways of tracking students program development processes. Some of them are one-to-one programming exercises, assisted programming sessions or student-led live coding sessions.

In one-to-one programming exercises, an instructor observes the students process while he/she is writing a program. Assisted programming session is pretty much the same with the one-to-one approach, but, it is applied to several students at the same time. Student-led live coding sessions refer to a selected student writing a program in front of his/her classmates, so that the process is visible to everyone. We find all of these methods intrusive, as they may effect students' behavior during program development. Indeed, it has been reported that some students were disturbed by personal observation of their work [7]. Chapter 9 further discusses these methods in related work and explain why we call them intrusive. The desired observation is that students internalize and use the process while they develop programs on their own.

An alternative approach to tracking program development is to embed the tracking ability into the development tool. Such a tool would need to record the development

process as students develop their programs. This process should be re-playable by the instructor, after students complete their development. Thus students can be observed without intervention to their work.

4. IMPLEMENTATION OF SCREEN-REPLAY

4.1. The Basic Approach

As stated in the motivation, in order gain insight into the impact of teaching programming with the HtDP design recipe, which teaches a process, it is necessary to track student programming sessions. For this, an approach that supports the following is proposed:

(i) the capture of programming sessions,
(ii) the observation of captured sessions,
(iii) the high level description of the process followed in a programming session,
(iv) the assessment of high level programming session descriptions.

The capture of programming sessions: In order to capture sessions consistently and objectively, programming sessions must be recorded automatically. All states of the development process, from beginning to end, must be captured. The recorded information must be persistent, so that it can be further observed and analyzed.

Observation of captured sessions: A captured session must be viewable. In other terms it must be re-playable. The presentation must identically recreate the actual

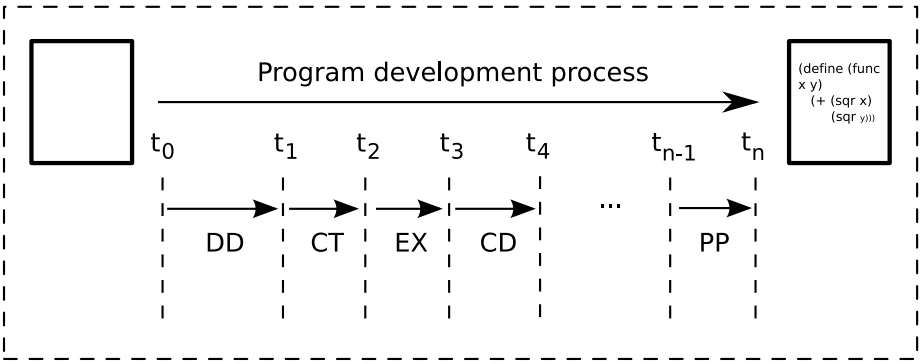

Figure 4.1. A programming session tagged with design step tags. The entire session starts at t_0 and terminates with t_n and sub-processes occur in sub-intervals.

programming session in terms of content, focus and time. An observer must be able to see the state of development at any given time and be able to browse the process.

High level description: A programming session can be considered as a process that consists of a sequence of subprocesses. In this case each subprocess would correspond to a HtDP design step. A developer may revisit a subprocess (design step) several times within the duration of a programming session. Each sub-process is associated with a time interval indicating it began and stopped working on that process. Each design step can be denoted with a tag. Tags can be used to annotate a programming session. We refer to these tags as *design step tags* and the process of associating such tags with specific time intervals as *programming session tagging* (or simply tagging session). The tagging of a programming session yields a sequence of design recipe tags that represent the entire process followed during development (see Figure 4.1).

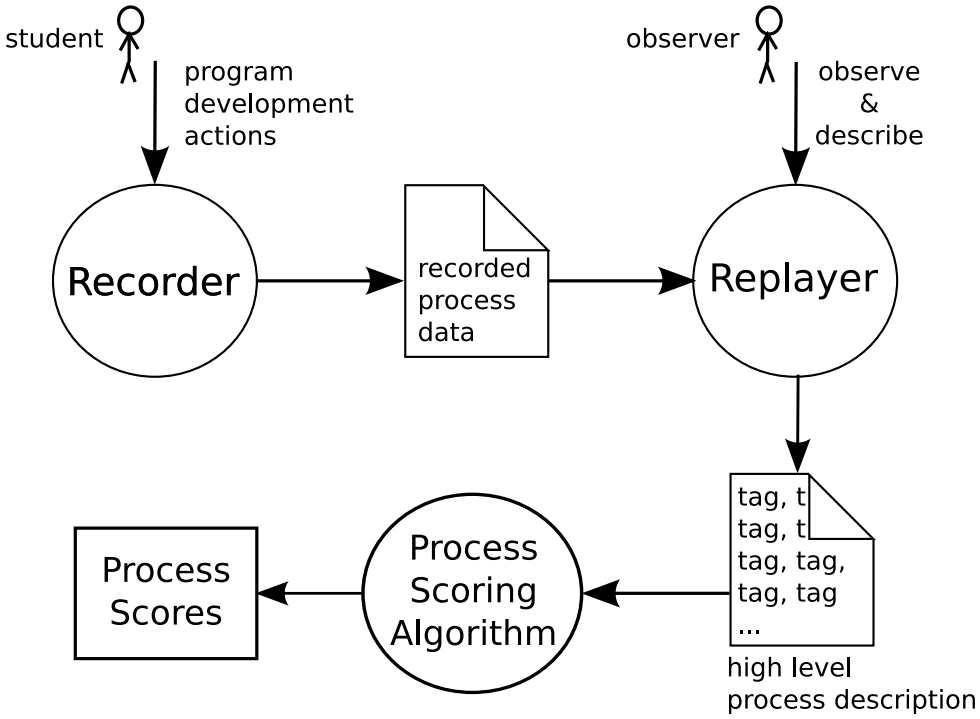

Figure 4.2. Overview of tracking and assessing programming session

Assessment: A high level process description can be evaluated by assigning it a score that represents how well it corresponds to the prescribed design recipe. We propose an algorithm that assigns a score to a given design recipe tag sequence. This algorithm rewards expected/prescribed design step transitions and penalizes those that

do not. This algorithm is concerned with relative ranking rather than assigning a fine grained scores that indicate levels of success.

A representation of proposed model is shown in Figure 4.2. As the model proposes, the *Recorder* automatically records every interaction of the student during the programming session. All recorded actions are saved in a file. This file, then, is used to view the recorded programming session and tag the programming session. Finally, the process scoring algorithm computes a process score for the design tag sequence.

4.2. Requirements

This section presents the requirements that based on the proposed model. In the remainder of this thesis the term *tag* should be understood as *design step tag*. The requirements are as follows:

(i) Capturing the programming session
- Every text related addition, deletion and modification must be saved.
- Each keystroke should be saved with the content, time and position data.

(ii) Observation of the development process
- The user should be able to select the file (i.e recorded session) to be viewed.
- The user should be able to view the session at the same speed as it occurred.
- Convenient browsing support for viewing sessions should be provided. The user should be able to pause, replay and navigate to an earlier time in session.
- If a design step tag sequence for a session was defined, it must be loaded along with session for viewing.

(iii) Describing the development process
- The user must be able to identify time intervals that correspond to a specific design step.
- The user must be able to identify time intervals that do not correspond to any design step (i.e. irrelevant activities).
- The user must be able to associate a description with a tag. This is for identifying the function that the tag belongs to, where the session includes

more than one function.

- The user should be able to see the defined tag sequence.
- The user must be able to delete an erroneous tag within the tag sequence.
- The user should be able to save a tag sequence.

(iv) User interface

- Recorded session must be displayed in a window.
- Appropriate tag buttons should be available for each design step.
- Appropriate control buttons for navigating a programming session should be available for each control function (i.e play, pause, play backwards).

(v) Persistence

- All interaction of a given session should be saved to a file.
- Each session must be stored in a separate file.
- Defined tag sequence should be saved to a file.
- Associations between sessions and tag sequences must be handled.

(vi) Scoring development process

- Points should be awarded for steps visited.
- Any premature visit of a step should be penalized.
- A penalty should be applied for each prerequisite step which was not visited.
- Each revisit of a previously visited step should be penalized.

4.3. Implementation

In order to track how students construct programs, we developed a system called Screen-Replay [29]. This system records how students develop their programs and allow evaluators to observe and identify the sequence of development activities taken during the program construction.

Screen-Replay mainly consists of two parts: Recorder and Replayer. It implements the requirements within the DrScheme environment. Scheme programming language is used for the implementation. The Recorder and Replayer are described in the following sections.

4.3.1. Recorder

The Recorder records all user interactions within the DrScheme's *Definitions* window, where programs are defined (see Figure 2.1). These interactions include insertions or deletions of any character. In other words, every time a user inserts a character into the editor or deletes a character from the editor, the Recorder saves this character along with some other additional information. This information is stored as a scheme structure, namely *action*, which is defined as in Figure 4.3.

```
;; action
;; timestamp (number): current time in seconds
;; operation (symbol): type of the operation
;;                     (can be 'insert or 'on-delete)
;; start (number)    : position of the cursor in the definitions
;;                     window at the time of operation
;; len (number)      : length of the action-content
;; content (string)  : the content of the action

(define-struct action (timestamp
                       operation
                       start
                       len
                       content) #:prefab)

;; For Example
(make-action 1240394142 'insert 0 1 "f")
```

Figure 4.3. The definition and example of the action structure

The above example is an action indicating that the user typed *f*, an insertion of length *1*, at position *0* of the definitions window, when the current time was *1240394142* in seconds. Position 0 is the starting position. Current time is generated using *current-seconds* function that returns the current time in seconds based on a platform-specific starting date and increases by one for each second that passes.

For every text insertion and deletion, the Recorder creates a corresponding action. These actions are accumulated in a buffer until the file is saved. When the file is saved, the buffer content (all the actions generated so far) is written to an *actions-file* with a ".rec" extension. The name of the actions-file is formed using the base file name of the program file. Subsequent actions are appended to the actions-file when the file is re-saved. In the case of a save-as operation, previous actions are copied from the current actions-file to the new actions-file with the new file name.

The Recorder catches keystrokes by extending the `definitions-text` with a `mixin`[5] . This mixin `augments`[6] the `insert` and `on-delete` methods. It uses a boolean flag that indicates the recording state of the current window. If the flag is true, it means we are in the recording state and every insertion of deletion is saved. If the flag is false generated actions are not saved. Using a separate flag for each editor window makes it possible to record actions in each window separately. In other words, a user may open more than one editor window and record each of them separately.

The Recorder is controlled using the recorder button placed on the DrScheme's main interface. This button is used to start and stop recording. The image on this button changes according to the recording state and informs the user about the current recording state (see Figure 4.4).

When the development process is recorded and saved as an actions-file, this file can be replayed to see the whole process exactly as it was written. The development process can also be annotated to obtain a high level description of the process that indicates the application order of the design steps.

4.3.2. Replayer

The Replayer has two main functions: (1) to replay the program construction and (2) to describe the high-level construction process with a tag sequence in terms of

[5]A mixin is a class extension that is parameterized with respect to its superclass.

[6]To augment means to override a method which is declared with pubment in a superclass. For further information, see [30]

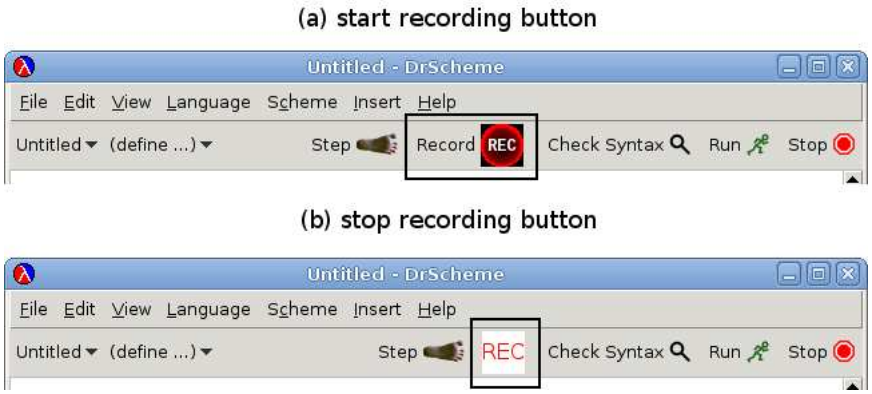

Figure 4.4. Start recording (a) and stop recording (b) buttons

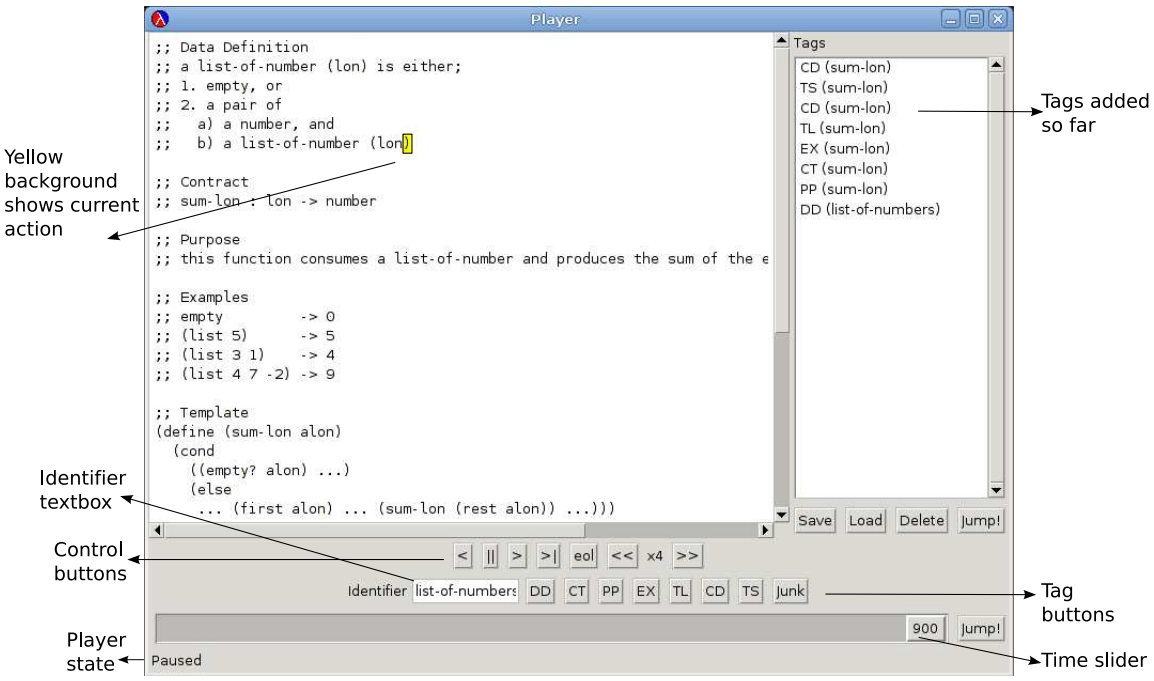

Figure 4.5. An overview of the Replayer

the HtDP methodology.

The Replayer allows the observer to see exactly how the program was constructed. While observing the construction process, the observer can annotate the programming session by associating time intervals with tags, so that each time interval corresponds to a design step.

4.3.2.1. Replaying. The Replayer replays the exact steps taken while the program was constructed. The observer can see each text insertion or deletion at the same speed of the construction process. Various controls enable more convenient navigation of the construction process:

- *Play*: Start playing actions
- *Pause*: Pause playing
- *Backwards*: Play backwards
- *Speed-Up/Down*: Change the play speed
- *Go-To-Next-Action*: Jump to next action without waiting
- *Time-Slider*: Directly navigate to a desired action.

A student may jump from one position to another during the programming session. For example, he/she can move to the data definition from the program code. Such jumps can make it difficult for the observer to follow the session. Additional features exist to assist the observer in such cases. For example, the Replayer automatically scrolls to the position within the program that is associated with the current action. This makes the location visible and enables the observer to follow the flow of construction. The location of the current action is also highlighted with a yellow background so that it is easily visible to the observer.

When a file is selected to be played, all actions in the associated actions-file are loaded into a scheme structure, namely *Tape*, which is defined as in Figure 4.6.

```
;; tape
;; actions (vector): contains the actions saved
;;                   by the Recorder
;; pointer (number): index of the current action

(define-struct tape (actions pointer) #:mutable)
```

Figure 4.6. The definition of the tape structure

When the *Play* button is clicked, a thread starts to play the actions in the Tape structure. To play an action is to insert/delete the content to/from the editor according to the timestamp and position information in it. For example, to play the following action

- (make-action 1240394142 'insert 0 1 "f")

will insert "f", a character with length 1, in the editors first position (position number 0). When playing backwards, the action operation is reversed: an *'insert* symbol is interpreted as *on-delete* and an *'on-delete* symbol is interpreted as *insert*.

The Replayer replays the construction at the same speed of the original construction. Scheme semaphores are used in order to make the Replayer wait while playing. The running thread is suspended until the semaphore becomes free. This semaphore is managed by a timer object, which is set to the difference between consecutive actions.

Consider that the example in Figure 2.3 is recorded using the Recorder. When the final product is inspected, it is clear that all design steps are present. The Replayer enables one to view the process that led to this product. The first column of Table 4.2 describes the students process. Replaying this session reveals that the student actually did not follow the design recipe sequence. It appears that the student attempted to fake the process. The student first implemented the code and then inserted the remaining required steps. The tracking process reveals the order of the application of design

recipe. It also shows how much time was spent on each step. The ability to observe such a process enables the instructor to discover deficiencies and provide more coherent help.

4.3.2.2. Tagging. Tagging allows the evaluator to describe the kinds of activities performed during the constructing of a program. A series of activities are associated with one the corresponding tags (see Table 4.1). During the tagging process the observer defines a sequence of tags as he/she observes the construction states. The interface includes buttons for each tag. The observer clicks on an appropriate tag button when the student moves from one state to another. For example, the observer clicks on the DD tag button, when the student finishes editing the data definition and moves to another design step.

Table 4.1. Design steps and associated tags

Design Step	Corresponding Tag
0. Data definition	DD
1. Contract	CT
2. Purpose	PP
3. Examples	EX
4. Template	TL
5. Code	CD
6. Tests	TS

If a student performs some activity that does not correspond to any design step, it is identified with the *Junk* tag. Junk may not be the best label, as the student may do something useful that is not directly meaningful to HtDP. For example, students may write a question or make a check list to assist themselves. On the other hand they may write something totally irrelevant, such as a note to the examiner (i.e. "Dear Professor, for God's sake, I don't want to fail."). In any case, the Junk tag should simply be interpreted as anything that is besides the tags defined in Table 4.1.

The ideal development sequence would be: [DD, CT, PP, EX, TL, CD, TS].

Naturally, one does not expect a perfect program construction. But, rather, hope to observe that the overall order of steps was followed.

```
;; tag
;; name (symbol)       : name of the tag
;; identifier (symbol): an identifier that the tag is tied to
;;                        can be the function name
;; end (number)        : the index of the last action in this tag
;;                        indicates the end point of this tag

(define-struct tag (name identifier end) #:prefab)

;; For Example
(make-tag 'DD 'list-of-numbers 65)
```

Figure 4.7. The definition and example of tag structure

Tags are stored in *Tag* structure (see Figure 4.7). The development process is represented with a sequence of tags, which are created by clicking on the appropriate tag-button. An identifier may be associated with a tag in order to further describe which function the activity is associated to, as a program may consist of several functions. For practical reasons, only the position of the tape-pointer at the end of the tag is stored. This makes reorganization of tags easier. When a new tag is generated, the Tagger saves this tag to its *tags-list* and displays it in the panel on the right side of the window.

Recall the program example in Section 2.4, which we assumed to be recorded using the Recorder. The second column of the Table 4.2 shows responses of the observer to the process of the student. The observer carefully tracks actions of the student and tags the session accordingly. At the end of the tagging process a tag-list, possibly as in the Figure 4.8, is generated. Tag-end positions may not be easily traceable from the given example, but they need to be shown in this example.

```
(list (make-tag 'CD 'sum-lon 158)
      (make-tag 'TS 'sum-lon 297)
      (make-tag 'CD 'sum-lon 303)
      (make-tag 'TL 'sum-lon 382)
      (make-tag 'EX 'sum-lon 484)
      (make-tag 'CT 'sum-lon 564)
      (make-tag 'PP 'sum-lon 574)
      (make-tag 'DD 'list-of-numbers 900)
      (make-tag 'JK 'none))
```

Figure 4.8. An example output of the tagging process

The above tag-list, generated from the tagging session, tells us that according to observer, actions between indices 0 and 158 are related to the program code (CD) for the sum-lon function. Similarly, actions between 159 and 297 are related to tests for the same function.

The application of design recipe was already visible by replaying the session, but having the tag-list in hand means much more than just replaying. First of all, once the tag-list is generated, there is no need to replay the session to see the process. It is sharable data, which can be sent to someone else for further observation. Tag-lists from different sessions of a student, or from different students can be used together to be analyzed. Even if the tagging process is not finished, tags generated so far can be saved and later loaded (for the same session) for further tagging. The Tagger also allows the observer to jump to a previously tagged position using the tag-list.

The tag-list, by itself, includes some information about the session and can be used for examining of the construction process. However, using both recorded actions and the tag-list together, a lot more information about the session can be extracted. The following subsection introduces the idea of "processed-tag" which enables more detailed investigation of a session.

4.4. Processing the Tags

While actions and tags are useful by themselves, merging these two sources of data provides a better insight to the students process. A tag, by itself, is actually a collection of actions. Therefore, it should represent characteristics of actions it contains. Using the time and position data already available in actions, tags can be extended with more information to generate a self contained analysis data. Processed-tag is defined as in Figure 4.9 to meet this requirement;

As described above, processed-tag includes much more information about the actions associated with a tag. Inspecting a processed-tag provides a summary of its associated actions, i.e duration, begin and end time, the segment in the code, etc.

It is possible to identify when the student switches between design steps. Inspecting these switches might reveal a common pattern in the application of the design recipe.

Another type of information that is possible to extract from processed-tags are the timings. Using processed-tags, it is possible to examine the time distribution among design steps.

It is possible to observe how the overall program progresses as well as individual segments. This information might provide insight into students' problem solving techniques.

Sessions can be divided into active or passive parts. Active parts are parts where the user interacts with the editor. Passive parts are the parts where the user does not interact with the editor and there is no information about what he/she is doing. The analysis of relations between these parts together with the segment switching information can provide more accurate information about the students' behavior.

```
;; processed-tag
;; step (symbol)          : the name of the tag
;; identifier (symbol)    : an identifier text
;; action-count (number) : total number of actions
;;                          contained in this tag
;; size (number)          : total length of actions
;;                          contained in this tag
;; start-time (number)    : time of the starting
;;                          action of this tag
;; end-time (number)      : time of the last
;;                          action of this tag
;; start-position(number): starting position of this
;;                          tag in the editor
;; end-position (number) : end position of this tag
;;                          in the editor
(define-struct processed-tag (step
                              identifier
                              record-count
                              size
                              start-time
                              end-time
                              start-position
                              end-position) #:prefab)
```

Figure 4.9. The definition of the processed-tag structure

Table 4.2. Students actions and observers responses in return

Student	Observer
Starts implementing the code for the sum-lon function.	Realizes that the student is implementing the code for the sum-lon function. Types an identifier (may be "sum-lon" for this case) or keeps it blank. Waits until the student switches to some other design step.
Finishes implementing the code. Starts implementing the tests.	Pushes CD button at the time student finishes the code implementation. Waits the student to finish the tests.
Finishes implementing the tests. Goes back to the code (he/she might get some errors. Tagger doesn't show it) and modifies some parts.	Pushes TS button at the time student finishes tests. Waits the student to finish code modification.
Finishes modifying the code. Starts writing template according to the code, then writes examples according to tests.	Pushes CD, then TL as the student finishes writing code and template, respectively. Waits him to finish examples.
Finishes writing examples. Writes contract and purpose for the function. Starts writing data definition for list-of-numbers.	Pushes EX, CT and PP buttons as the student finishes writing examples, contract and purpose, respectively. Updates the identifier while the student is writing data definition for the list-of-number. Waits the student to finish the data definition.
Finishes writing data definition.	Pushes DD button as the student finishes writing the data definition.
Writes his/her name and ID number.	Pushes JK button (as this is an irrelevant information for the analysis) as the student finishes writing identification information.

5. PROCESS SCORING ALGORITHM

Replaying the recorded programming sessions already revealed the programming behaviors of students. But, still, we do not have an answer to the question of "how good a student followed the design recipe". In other words, we need a concrete value to indicate the degree of "following the design recipe". Such a value will serve as a comparison criteria. This section introduces a process scoring algorithm that produces similarity scores for tagged programming sessions.

5.1. Assessing the Design

The Design Recipe (as explained in Section 2.4) is a check-list for students and it has an order that must be followed during the program construction. In the best case, it should appear in the following order: [DD, CT, PP, EX, TL, CD, TS]. But we may not expect a student to fully follow this exact pattern, and it is not true to say that a student can not benefit from the design recipe if his/her design pattern does not exactly match with the suggested one.

A student may start with the data definition, then jump to examples, and then write the code. It's obvious that such a pattern does not meet all the requirements of the design recipe, but the students intention to follow it is still observable. Thus, we may not assess the students design pattern with a binary grading scheme (i.e 1 for exactly the same pattern, and 0 for others). Instead, we need to determine the similarity of the students pattern with the suggested one. To do that, we used a custom similarity algorithm that calculates a similarity score for an input design pattern.

Following sections discuss the "similarity" notion and the similarity scoring algorithm that we use to assess students design patterns.

5.2. Classical String Similarity Algorithms

There are several string similarity (or edit distance) algorithms that are used to measure how similar two strings are. These algorithms have been used in different application areas like computational biology, signal processing, text retrieval, etc [31]. One of the best know similarity algorithm is the Levenshtein [32] distance algorithm. It calculates the least number of edit operations that are necessary to modify one string to obtain another string [33]. Allowable operations are insertion, deletion, and substitution. Each operation has a cost of 1. The aim is to start with the first string and use the allowable operations to transform the characters until the second string is reached. The distance between the first and the second string is the minimum operation cost. As an example, suppose that we are trying to measure the distance between strings "kitten" and "sitting" [34].

(i) kitten (substitute "k" with "s")
(ii) sitten (substitute "e" with "i")
(iii) sittin (add "g" at the end of the string)
(iv) sitting (total cost 3)

Considering that a design pattern of a students is actually a series of symbols, we can apply the same algorithm to measure the distance between the suggested design recipe and the design pattern of a student. As an example, suppose that we have two different design patterns from 2 students: For the first student [DD, CT, EX, PP, TS, CD] and for the second student [CD, TS, DD, CD, EX, TL]. Using the same algorithm above, we will try to measure the distance between the suggested pattern [DD, CT, PP, EX, TL, CD, TS] and given example patterns. As a result we will produce distance scores for each student. The smaller distance score will mean that the design pattern of that student is more similar to the suggested pattern then the one with the larger distance score. If the algorithm produces the same score for both design patterns, this means that they are equally similar to the suggested one.

For student 1:

- (DD, CT, EX ,PP, CT, CD) (substitute EX with PP)
- (DD, CT, PP, PP, CT, CD) (substitute the second PP with EX)
- (DD, CT, PP, EX, CT, CD) (substitute the second CT with TL)
- (DD, CT, PP, EX, TL, CD) (add TS at the end of the list)
- (DD, CT, PP, EX, TL, CD, TS) (total cost is 4)

For student 2:

- (CD, CT, CD, EX, TL, DD, CD) (substitute the first CD with DD)
- (DD, CT, CD, EX, TL, DD, CD) (substitute the first CD with PP)
- (DD, CT, PP, EX, TL, DD, CD) (delete the second DD)
- (DD, CT, PP, EX, TL, CD) (add TS at the end of the list)
- (DD, CT, PP, EX, TL, CD, TS) (total cost is 4)

As illustrated above, the distance algorithm produced same distance scores for both design patterns, which indicates these patterns are equally similar to the suggested one. But, the algorithm did not pay any attention to the order of symbols in the list. The design recipe approach, however, is only beneficial when the order is applied as suggested. Let us analyze these two patterns in more detail and decide whether they are really equally similar to the suggested pattern.

Student-1 started the process by defining the data (DD) that he will use for this program. Then, he stated the input and output types (CT) of the program, most probably using the data he defined in the previous step. To be able to figure out how the program should behave, he gave some examples (EX) according to the input-output types. He stated a purpose statement (PP) for the program. He revised his contract (CT). And finally, using the data definition, function contract, purpose and examples he wrote the code (CD) of the program.

Student-2 started the process by writing the code (CD). Since he did not know the input and output types, he stopped writing the code and continued by defining the function contract (CT). Then, using the function contract he modified the code (CD). But, still, he had no idea how the program should behave for different types of inputs. By giving some examples (EX) he tried to understand the behavior of the program. After that, he wrote a template for the program (TL). Since the template of the program is decided by the data that is used, a need to a data definition (DD) emerged. Lastly, he revisited the code (CD) with a better understanding of the program.

A detailed analysis of two design patterns revealed that student-1 actually followed a similar path to the suggested pattern and benefited more from the design recipe. Student-2, however, directly jumped into the code without going through the required design steps, and had to go back to previous steps in order to have a better understanding of the program.

This illustration showed that rather than a classical string similarity algorithm, we needed to use a custom algorithm that pays special attention to the order of design steps to be able to calculate more accurate process scores. Thus, we developed our own algorithm to assess design patterns of students. The following section explains our process scoring algorithm in more detail.

5.3. Our Scoring Algorithm

HtDP requires students to go through (1) all design steps (2) in the suggested order and (3) to make them properly as defined. The third requirement actually means "do not proceed until you properly finish a design step". Because any deficiency in a step will inherit to other consecutive steps and may result in erroneous programs. These three requirements are transfered to our algorithm as follows:

(i) The student should get points only for steps he/she visited.
(ii) Any premature attempt to write a step should be penalized.
(iii) Each revisit of a previously visited step should be penalized.

According to our algorithm, each design step, 7 in total, have the same score (14.28), which yields a total of 100 points, if every step is done properly. This means that each step missing from the design recipe costs students 14.28 points.

- [DD, CT, PP, EX, TL, CD, TS] → 100 pts
- [DD, CT, PP] → 42.84 pts

On the other hand, to get the full point (14.28) from any design step, a student must visit all the prerequisite steps. For example, to get the full point for CT, the student should have previously visited DD. Otherwise, he/she can only acquire half of the points (7.14). The rationale is straightforward: the student did not benefited from a proper data definition, and made a premature attempt to write a purpose statement. Similarly, before writing CD, 5 steps (DD, CT, PP, EX and TL) must be visited. For each skipped step, 1/6 of the full point (2.38) will be reduced.

- [DD] → 14.28 pts
- [CT] → 7.14 pts
- [DD, PP] → 14.28 + 9.52 = 23.80 pts
- [DD, PP, CT, EX, CD] → 14.28 + 9.52 + 14.28 + 14.28 + 11.9 = 64.26 pts

The third rule proposes to reduce points for each revisit of a step. On one hand, revisiting a step is not a bad thing, since a modification may be done with the intention of increasing the quality of that design step. On the other hand, it shows that previous attempts were not sufficient, as a modification is needed. Therefore, we reduce only 1 point for each revisit operation.

- [DD, CT, PP, EX, TL, CD, TS] → 100 pts
- [DD, CT, PP, EX, TL, CD, TS, CD, TS] → 98 pts
- [DD, PP, CT, EX, CD] → 64.26 pts
- [DD, PP, CT, EX, CD, DD, CD, EX, CD] → 60.26 pts

```
score ← 100
visitedSoFar ← empty
for each step in pattern do
  if visitedBefore?(step, visitedSoFar) then
    score ← score − 1
  else
    score ← score - (pointsForThisStep(step) * numberOfSkippedSteps(step,
    visitedSoFar))
  end if
  visitedSoFar ← visitedSoFar.append(step)
end for
score ← score − (14.28 ∗ numberOfMissingSteps(pattern))
return round(score)
```

Figure 5.1. Pseudo-code of the process scoring algorithm

Our algorithm implements these requirements in a different manner. First, it assigns students the initial perfect score (100 pts), and then reduces this score according to the rules above (see Figure 5.1). Maximum score (100 pts) is only granted for the suggested pattern itself. For the minimum score, however, there is no lower-bound. The algorithm may produce a negative score, if a pattern includes too many revisited steps.

This algorithm is not intended to produce any result that is the exact answer (if such thing exists) of how good a student followed the design recipe. Instead, it produces similarity values that are meaningful in relative basis. That means, it produces higher scores for patterns that obey the rules of the design recipe, and produces lower scores for patterns that follow a random order instead.

The evaluation of the algorithm itself is another question. It is not possible to write a checker for this algorithm to test its accuracy. We can check, however, whether produced results are relatively accurate according to HtDP instructors. In order to do that, we asked 5 different instructors (lecturers and teaching assistants), who are

Table 5.1. Evaluation of the process scoring algorithm

Patterns / Instructors	I#1	I#2	I#3	I#4	I#5	Alg.
(DD PP DD PP DD EX CD PP EX TL)	3	3	3	3	3	3 (51)
(DD CT DD CT CD PP EX TL DD TL CD)	4	4	4	5	5	5 (74)
(DD CT)	2	1	2	2	2	2 (29)
(DD CD DD CD DD CD DD CD)	1	2	1	1	1	1 (13)
(DD CT PP EX TL CD TS CD TL DD CT DD CT PP CT EX CT EX DD EX TL DD EX DD EX TS DD TS DD TL DD TL CD DD CD PP CD DD)	5	5	5	4	4	4 (69)

familiar with the HtDP approach but did not have prior knowledge about our algorithm, to grade 5 patterns from 1 to 5 (5 indicates most similar to the suggested pattern, 1 indicates least similar). These patterns were randomly chosen from the exam data that we used for our experiment. Table 5.1 shows these patterns along with grades from 5 instructors and scores generated by our algorithm. Our algorithm produced very similar results to instructors.

Our similarity scoring algorithm successfully generated accurate results that indicated how good students followed the design recipe. We conducted an experiment using resulting scores together with exam grades to find out whether students who follow suggested guidelines were more successful than the others.

6. EXPERIMENT

In order to answer the questions stated in the introduction section, we conducted a small experiment. This section presents the experimental setup and methods used to evaluate our results.

6.1. Experimental Questions

This experiment has been done in order to answer the following questions:

- Do students apply the design guidelines we teach, when they develop programs on their own?
- Are students, who follow the suggested guidelines, more successful than the others?
- If not, can we identify patterns or approaches used by successful students?

6.2. Participants

Participants of this experiment were students who took the 4th live-programming exam of Comp149-150/HtDP course in the 2008-2009 academic year. A total of 77 students participated in the exam. The exam was conducted in two different sessions in order to fit students in computer labs. We recieved 57 submissions from session 1 and 20 submissions from session 2. Out of 77 submissions, 61 were used for the experiment. Table 6.1 shows the demographic information of the participants.

Table 6.1. Demographic information of exam participants

Department / Gender	Male	Female
Computer Science	18	9
Financial Mathematics	9	13
Business Informatics	7	5

6.3. Tasks

This exam was the last one among 4 live-programming exams and it covered almost all the topics presented in Section 3.2. It was done in two different sessions. Each session consisted of one question that was actually very similar to the other, but used different domain terminology. Thereby, any possibility that students in the afternoon session (session 2) may take any solutions from students in the morning session (session 1) was prevented. Figure 6.1 presents the questions asked for each section.

Session 1

Each particular field of any modern science is developed by its literature consisting of publications. When a paper is written, it is a very important matter that its author(s) cites related studies which are already published. However, given a literature, it is rare to find a cycle of citations within the literature. For instance a cycle of paper citations would be formed if A cites B, B cites C, and C in turn cites A.

A publication is defined as a Scheme structure with its properties: title (a string), authors (a list of strings) and a list of cited articles identified by their titles.

Design and implement a program, given a publication title and a literature, it starts to explore the literature reading through the references of the given paper to check whether there is a cycle of citations or not. If there is a cycle, it returns names of authors of all publication in the cycle, otherwise False.

Session 2

A Website is a list of webpages. Each page is a Scheme structure of properties: page title (a string), editors (list of strings) and links to other webpages in the website.

Your program starts crawling at a given Web page and explores other Web pages following the links. For instance, when it visits following sequence of links:

A → B → C → D → B

then it should discover that there is a loop of hyper links formed by sequence of B → C → D → B and it should report the editors of these web pages (B, C, D, B).

Design and implement a program given a Web page and the Website, it returns editors of pages which form the very first loop of Web links it discovers. If there is none, it should return a False.

Figure 6.1. Exam questions for session 1 and session 2

Students had 1 hour to solve the question. Between two sessions we needed 15 minutes break to configure the lab for the next exam. Students sent their solutions to

the submission server along with the ".rec" files, which were used to monitor students programming activities.

6.4. Monitoring

Monitoring was done using the tracker software that we developed. During the exam, students developed their programs using DrScheme, which included the Screen-Replay tool as a module. The tool was pre-configured to start recording as soon as DrScheme was initiated. We also hided the "Stop recording" button, because we did not want students to stop recording until the end of the exam. Recording automatically stopped when DrScheme was closed.

During the exam, every insertion and deletion to the editor was recorded. This information was saved in a ".rec" file. Students sent these files along with their solutions (".scm" files). We realized later on that some of our students failed to submit the ".rec" file. Some others used separate editor windows to develop their programs and, at the end of the exam, merged all functions into one file. Thus, ".rec" files of those students were only including actions of final copy-paste operations. Because of these problems, we excluded those submissions from the experiment data.

6.5. Grading the Exam

The common way of evaluating exams among HtDP instructors is to grade each design step separately. Our investigation of discussions about grading in the PLT-edu[7] mailing list also verifies this observation. To grade the live-exam, we also used the same approach.

Questions in both sessions in our exam required a main function and a couple of helper functions in order to be solved correctly. We divided those functions into two different groups as "main function" and "helper functions". The number of helper functions may vary, since some students prefer to write anonymous helper functions

[7]The mailing list for educators who participate in the TeachScheme! Project

Table 6.2. Grading each function using the scala

Std	Main Function							Helper Function # 1							Helper Function # 2						
Id	DD	CT	PP	EX	TL	CD	TS	DD	CT	PP	EX	TL	CD	TS	DD	CT	PP	EX	TL	CD	TS
1	3	3	3	3	3	2	0	-							-						
2	3	1	1	3	0	1	1	-	2	3	1	2	3	1	-						
3	3	2	3	0	2	2	0	-	3	3	0	1	3	0	-	3	2	0	2	3	0

(functions without a name) using lambda-expressions which is defined within other functions. We observed 1 to 6 helper functions among the solutions for this live exam. The first step of grading is illustrated in Table 6.2 for 3 example students. As shown in the table, each function is graded separately according to steps of the design recipe. For simplicity, each design recipe step is first graded using this scala:

- does not exists or is not acceptable: 0
- insufficient: 1
- intermediate: 2
- good: 3

For this example illustration, only 2 helper functions are introduced (as explained above, 1 to 6 helper functions were observed from solutions of students). According to this example, student-1 only has the main function, student-2 has the main function and one helper function, and student-3 has the main function and 2 helper functions. Data definitions (DD) of helper functions are marked as "-" in the table, since the problem statement for each question only asked for a single data definition. During the exam, students wrote a data definition for the main function and used it for helper functions, as well. Therefore, DD was only graded for the main function.

After grading each function using the scale, scales are converted to real grades that are over 100. Table 6.3 shows this conversion using the previous example. Conversion criteria may change according to the hardness of the questions. It is decided as follows

Table 6.3. Scale grades converted to number grades over 100

Std	Main Function							Helper Function # 1							Helper Function # 2						
Id	DD	CT	PP	EX	TL	CD	TS	DD	CT	PP	EX	TL	CD	TS	DD	CT	PP	EX	TL	CD	TS
1	100	100	100	100	100	75	0	-							-						
2	100	40	40	100	0	40	40	-	75	100	40	75	100	40	-						
3	100	75	100	0	75	75	0	-	100	100	0	40	100	0	-	100	75	0	75	100	0

for this specific exam:

- $0 \rightarrow 0$
- $1 \rightarrow 40$
- $2 \rightarrow 75$
- $3 \rightarrow 100$

Calculation of grade for each function is done by joining grades for design steps together. But, every design step has its own weight and it changes according to the function type. Table 6.4 presents these weights for both function types. Since the problem statement only required a single data definition and students defined it for the main function, we did not use same weights for main and helper functions. The weight for DD is distributed to other design steps for helper functions. Using these weights, grades in Table 6.3 are recalculated as shown in Table 6.5.

Grades of design steps for each function are summed to calculate the individual function grades. As solutions included different numbers of helper functions, we calculated the average grade of all helper functions and used this value to calculate the final grade. Final grade is the sum of 60 per cent of the main function grade and 40 per cent of the helper function average grade, as shown in Table 6.6.

This experiment is conducted to collect two kinds of data. Using Screen-Replay

Table 6.4. Percentages of design step weights for main and helper functions

Design Step	Main function	Helper functions
DD	20	-
CT	10	15
PP	5	10
EX	10	15
TL	15	15
CD	30	30
TS	10	15

Table 6.5. Application of design step weights

Std	Main Function							Helper Function # 1							Helper Function # 2						
Id	DD	CT	PP	EX	TL	CD	TS	DD	CT	PP	EX	TL	CD	TS	DD	CT	PP	EX	TL	CD	TS
1	20	10	5	10	15	22.5	0	-							-						
2	20	4	2	10	0	12	4	-	11.25	10	6	11.25	30	6	-						
3	20	7.5	5	0	11.25	22.5	0	-	15	10	0	6	30	0	-	15	7.5	0	11.25	30	0

Table 6.6. Calculation of the final grade

Std Id	Main Function Grade	Helper Function Average Grade	Final Grade
1	82.5	0	49.5
2	52	74.5	61
3	66.25	62.38	64.7

tool, programming activities of students are recorded, replayed and annotated. Process scoring algorithm is used to produce similarity scores. Therefore, we obtained values that indicate how good our students followed the design recipe. Grades, generated using the method explained above, indicate the success of students in the exam. In the next section, we will combine these two types of data together to examine the impact of using the design recipe.

7. EVALUATION

The experiment we conducted mainly produced two types of outputs: process scores that indicate how well students followed the design recipe and exam grades that indicate the success of students in the exam. These outputs were examined together in order to answer the questions stated in the introduction. Process scores and exam grades are calculated separately for each function type (main and helper functions), so that we can examine these questions on a function bases. Following sections show the evaluation for each question separately.

7.1. Question 1

The first question was: "Do students apply the design guidelines we teach, when they develop programs on their own?". To answer this question, it is enough to investigate the design step tags produced by the Screen-Replay tool.

Design step tags for the main function indicate that students applied the design recipe in different orders, but the common intention was to follow the suggested order. All of the 61 students started with the data definition (DD) and 45 of them continued with the function contract (CT). After that 30 students wrote the purpose statements (PP), while most of the others jumped back to the data definition.

A different situation is observed for helper functions. A total of 91 helper functions were written by 42 students. The rest, 19 students, did not write any helper functions. For helper functions, students were expected to start the development process with the function contract (CT), since the programming task required a single data definition and students already defined it for the main function. But, only 24 out of 91 helper functions started with the function contract. 61 helper functions are written starting with the program code. As Screen-Replay revealed, such students copied similar code segments from the text book and pasted into their editor. Only 27 of these students continued with the function contract, while others jump to other

design steps.

As a result, observations showed that students followed design recipes better when writing the main function. Process scores also indicate similar results. The process score average for the main function is 66, while it is 57 for helper functions. The median value is, again, higher for the main function. It is 73 for main function and 63 for the helper functions.

7.2. Question 2

The second question was: "Are students, who follow the suggested guidelines, more successful than the others?". In other words, do high process scores constitute to high grades from the exam, or vice versa. To examine this, process scores and exam grades are divided into sub categories.

Exam grades are separated into three categories as high grades, average grades and low grades. Separation points are determined according to the academic regulations of Istanbul Bilgi University [35]. Table 7.1 shows how our categories map to the university regulation.

Process scores are divided into two categories. The separation point is determined by the median value. Process scores higher than the median value are grouped as "high process scores" and process scores lower than the median value are grouped as "low process scores". Number of students in these groups were slightly different for main and helper functions, since the median values were different.

Results are examined separately for each function type. For the main function, median value of process scores was 73. There were 6 students with this value. These students were added into the "low process score" group and 74 was taken as the separation point (see Table 7.2). As shown in Table 7.4, 12 students acquired high grades for the main function and 8 of them were students who had high process scores. For students with average grade, number of students with high process score (11) is, again,

Table 7.1. Categorization of the exam grades

Letter Grade	Definition	Category	Main Function # of students	Helper Functions # of students
A	Excellent	High Grade	12	10
A-	Excellent			
B+	Good	Average Grade	19	10
B	Good			
B-	Good			
C+	Average			
C	Average			
C-	Pass on probation	Low Grade	30	41
D+	Pass on probation			
D	Pass on probation			
F	Fail			

Table 7.2. Process score categorization for the main function

Category	Criteria	Number of Students
High process score	score ≥ 74	29
Low process score	score < 74	32

Table 7.3. Process score categorization for helper functions

Category	Criteria	Number of Students
High process score	score ≥ 45	30
Low process score	score < 45	31

more than the number of students with low process score (8). For low grades, however, students with low process scores are much more higher than high process scores. As a result, main function data indicated that, students with higher process scores acquired higher grades than the students with low process scores.

Table 7.4. Process scores vs. exam grades for main function

Process score \ Exam grade	High grade	Average grade	Low Grade
High process score	8	11	10
Low process score	4	8	20

The results for helper functions are similar to the main function. For helper functions, 10 students acquired high grades and 9 of them were the ones with high process scores. For average grade students, number of students with high process scores is, again, higher than the ones with low process scores (7 and 3). 41 students acquired low grades and 27 of them had low process scores. As a result, students with high process scores were more than the others for high and average grades, and less than the others for low grades.

Table 7.5. Process scores vs. exam grades for helper functions

Process score \ Exam grade	High grade	Average grade	Low grade
High process score	9	7	14
Low process score	1	3	27

As mentioned earlier, 19 students did not attempt to write any helper functions. There were also some students that started to write but then deleted the helper functions. Therefore, the data for the helper functions included zero values for both process scores and helper function grades. Excluding zero values, helper function results are recalculated for 38 students (see Table 7.6). For students with high grades, results were similar to the previous calculations: number of students with high process scores were higher than the ones with low process scores. For students with average and low grades, however, results were different. There were less students with high process scores in average grade group and more students in low grade group.

Table 7.6. Process scores vs. exam grades for helper functions (excluding zero values)

Process score \ Exam grade	High grade	Average grade	Low grade
High process score	7	2	10
Low process score	3	8	8

As explained in Section 6.5, the grade of each function consists of the grade of design steps associated with that function. In other words, all the design steps have an effect on the function grade. When we say for example "high process scores constituted high exam grades" it actually means "following the design recipe in the suggested order constituted high quality design steps". One may wonder whether following the design recipe in the suggested order constitutes more quality program code. Table 7.7 shows the results for the main function data, where the grades were calculated from the program code only, other design steps were excluded. The number of students for each category is similar to the previous results and indicate high process scores constitute to more quality program code. In Table 7.8, helper function process scores (excluding zero values) are mapped into helper function code grades. Again, other design steps were excluded and just the program code is graded. For this data, the number of students for high and low process scores are not much different for each category. Students with low process scores were slightly higher for students with high grade, and vice versa for students with average grades, therefore, did not constitute to significant results.

Table 7.7. Process scores vs. exam code grades for main function

Process score \ Exam code grade	High grade	Average grade	Low grade
High process score	10	16	3
Low process score	6	12	14

Table 7.8. Process scores vs. exam code grades for helper functions (excluding 0's)

Process score \ Exam code grade	High grade	Average grade	Low grade
High process score	9	8	2
Low process score	11	6	2

7.3. Question 3

The third question was: "Can we identify patterns or approaches (in terms of design recipe) used by successful students?". To answer this question, design step tags (high level process descriptions) are further investigated. For a detailed investigation, design step tags are categorized in three groups. Categories are determined according to the order of design steps that the students follow during the development process. Description of these categories are as follows:

(i) A *proper step* means that the student already visited all of the prerequisite design steps.
(ii) A *revisited step* means that the student already visited this design step.
(iii) A *premature step* means that the student has not yet visited the prerequisite design steps. As long as prerequisite steps are not completed, they will be classified as premature steps even though they may appear as revisited.

For example, lets consider the tag sequence [DD, CT, EX, TL, EX, PP, CD, EX, DD, TS]. Here, tags in positions 3, 4 and 5 (EX, TL and EX) are premature steps and tags in positions 8 and 9 (EX and DD) are revisited steps. The maximum number of proper tags can be at most 7. For the revisited and premature tags, however, there is no upper bound. A design tag sequence is considered better, when the number of proper tags are closer to 7, and the number of premature tags are closer to 0.

Table 7.9. Exam grade vs. design step category

Exam grade \ Design step category	Proper	Revisited	Premature
High grade	6.17	15.42	1
Average grade	5.63	9.58	1.53
Low grade	4.17	9.37	1.23

For the main function data, each tag sequence is examined and the average number of proper, revisited and premature tags are calculated. Table 7.9 shows these averages grouped by the exam grade categories. It turned out that students with high

exam grades (successful students) have the highest average for the proper step category (6.17) and the lowest average for the premature step category (1). What is interesting is that, successful students are the ones who have the highest number of revisited steps (15.42), and it is much higher than the average and low exam grade students. This may indicate that successful students attempt to perfect their solutions.

As stated earlier, instead of following the suggested order of the design recipe, some students prefer to jump from one design step to another, which is not the successor step. Detailed investigation of design steps also revealed that some transactions (a jump from one design step to another) occur more than the others during the development process. As Table 7.10 shows, frequently used transactions are from DD to CT (8.58 per cent), from DD to EX (8.48) and from TL to CD (7.37). For successful students, however, the percentages of these transactions are slightly different. Successful students mostly jumped from TL to CD (9.58), from DD to EX (8.43) and from DD to CT (7.66).

As a result, during the development process, successful students (students with high exam grades) frequently jump back and modify previously visited steps. They mostly jump from TL to CD, which means that they follow the suggested approach by first writing the function template and then moving to program code.

Table 7.10. Frequently used transactions by all students (over 5 per cent)

From	To	Percentage
DD	CT	8.58
DD	EX	8.48
TL	CD	7.37
EX	DD	7.16
CT	PP	5.45
CD	DD	5.35
DD	CD	5.25
EX	TL	5.25

Table 7.11. Frequently used transactions by successfull students (over 5 per cent)

From	To	Percentage
TL	CD	9.58
DD	EX	8.43
DD	CT	7.66
EX	DD	7.28
EX	TL	5.36

8. DISCUSSION

8.1. Screen-Replay

The fuzzy nature of the design recipe makes it hard to automatically detect the design segments. Students apply it in different orders and in different forms. Steps other than code and tests do not have formal definitions. Some heuristics may be developed, but they can hardly ensure a precise tagging. Therefore, instead of automation we preferred to support the observer with helper functionalities in order to reduce the time and effort required for tagging.

To make tagging easier *go-to-next-record* and *go-to-end-of-the-current-line* buttons are added to the Tagger. The former enables the reviewer to jump to the next action without waiting the action to be occurred. And the latter enables the reviewer to jump to the action that takes place at the end of the current line. Since students change the current line when starting to write a new design step, using this button makes tagging easier.

Another feature that assists the observer is the jump detection function. This function pauses the playing process and warns the observer when the user is about to jump 3 lines above or below from the current line. The observer, then, may put a new tag or continue playing. According to our observations, one or two line jumps mostly appear within the same tag. Therefore, we preferred to warn the observer every time a 3-line-jump occur.

An interesting side effect of the Screen-Replay tool is related to plagiarism, which can be used during analysis. Detecting plagiarism was not a design decision for Screen-Replay, but an analysis of the actions-file helps us to detect plagiarism. For example, a student may copy-paste someone else's code. Since the Recorder generates a new action for each keystroke, copy-paste operations end up with actions that has a length greater than 1.

We compared recorded sessions with source codes to verify that recorded sessions would build the exact source code. All sessions were successfully regenerated from the recorded files, with the exception of regions that were commented out with boxes[8] , since this feature has not yet been implemented.

8.2. Process Scoring Algorithm

Scores generated by our similarity scoring algorithm is an issue to be discussed. This algorithm is not intended to produce any result that is the exact answer (if such thing exists) of how good a student followed the design recipe. Instead it produces similarity values that are meaningful in relative basis. For our experiment relatively accurate results are sufficient, since we are interested in the statistical relationship between grades and process scores.

Another question is the evaluation of the algorithm itself. How can we test the accuracy of such an algorithm? To the best of our knowledge, this is the first attempt to develop such an algorithm. Therefore, to test the accuracy, we used the only available source we have: HtDP instructors in our department. Instructors are asked to rank 5 randomly selected design patterns from 1 to 5, and the result produced by our algorithm was not distinguishable among the ones produced by instructors.

8.3. Experiment

Normally, in a *controlled experiment*, experimental units (students in this case) are divided into two different group: control and treatment [36]. Then the *independent variable* is applied to the treatment group to test the effect of that variable. In our case, the independent variable is the "use of design recipe". This means that only one of the groups should use the design recipe in order to test its direct effect on the exam grades. In a introductory programming course, however, this method is not appropriate. All students of the course (unless it is design for a special purpose), should be taught using same tools and methods.

[8]DrScheme allows users to comment out regions with a box snip.

Measurements are usually subject to variation. The experiment we conducted can be replicated to eliminate these variations and obtain more confident results.

We use a simple model to observe the effect of design recipe on the exam grades. Using a more complex model may lead to a better understanding of underlying relations between these two data sets.

Our preliminary observations indicated that high process scores constitutes high exam grades for most of the cases. However, we strongly believe that there is a more strong relationship between these two data sets. We identified some problems in our experimental setup, which, we believe, reduced the relationship between process scores and exam grades. First of all, the solution for the exam required 6 helper functions to be properly solved. During the exam, instead of concentrating on a single function at a time, students jumped across functions to correct any mistake they realized. This behavior dilute the strength of the design recipe, therefore, reduce students process scores. Instead of a single task which requires 6 helper functions, we could set two different tasks that require at most 2-3 helper functions and give the second task when students finish the first one. Second, students wrote (anonymous) functions within other functions, which makes function templates inappropriate, and leads to a lower exam grade.

9. RELATED WORK

To the best of our knowledge, there is no software package dealing with the analysis of code/editing sequences in the way Screen-Replay does. This section rather reports approaches that intend to increase both product and process quality of students in programming classes.

In [37], authors report on a controlled experiment to evaluate whether students using continuous testing are more successful in completing programming assignments. As the source code is edited, continuous testing uses excess cycles on a developer's workstation. It continuously run regression tests in the background against the current version of the code, providing feedback about test failures. Their tool aims to give extra feedback during the programming session and improve the productivity of developers. Experimental results indicated that students using continuous testing were more likely to complete the assignment by the deadline. It appears that their efforts are on final product quality rather than the programming process.

In their case study [1], instructors from Tübingen and Freiburg Universities report the development of their introductory programming course. For their first-year programming course, they adopted the tools developed by the TeachScheme! project. In addition, they supervise their students closely with assisted programming sessions on weekly basis. During assisted programming sessions, students solve a set of exercises under the supervision of a doctoral student assisted by one or two teaching assistants (TAs) to ensure that the students follow the design recipes. Authors report that their students not only performed well on exams, they were also able to transfer their knowledge to other programming languages and IDEs. However, we observed that some students perform poorly (some even could not do anything) when they are watched "over their shoulders" during programming sessions. Such students perform well when they study in environments, where they feel comfortable. As stated in the same study, nearly 15 per cent of the students did not even try to solve the programming assignments during assisted programming sessions. We can not say that this is

caused by the same reason, but, further analysis can be done, and the sessions of such students can be observed later using tool support.

The same study also points out that many students avoided asking TAs for help during the session, as they either expected that TAs were not allowed to provide concrete help or they even believed that asking for help was a form of cheating. As reported, the perception of assisted programming changed during the semester as TAs not only provided help upon request, but also helped pro-actively as they noticed students having problems. This approach is helpful for students, who hesitate asking questions. The point is, how do we find out that a student is experiencing a problem applying the design recipe without constantly watching his/her session? As we have already experienced, students' main concern is to have the final running code before the time finishes. Thus, they escape from applying the design recipe and focus back on the code using the programming-by-tinkering method, as soon as they stay uncontrolled. Furthermore, assisting students during programming sessions does not mean that they apply design recipe in exams. One may not attribute the success of students to the success of design recipe, without tracking their process during exams.

Another study [38] points out the importance of exposing the process of development of the solution rather than just presenting the final state of the program. They propose "live coding" as an active learning process. Since instructors do not commit same errors students generally do, they suggest the student-led live coding (where the student writes the code in front of his/her classmates) rather than the instructor-led live coding (in which the instructor writes the code in front of students). Our experiences show that, especially in the first few weeks, students should program by themselves and learn from their mistakes. Interrupting as they make mistakes means taking their chance of solving the problem by themselves and, therefore, learning the importance of design recipe.

Exposing errors of a student in front of his/her classmates might also damage the motivation of other students and lead them to hold back and not participate. Instead, project submissions of students can be replayed without showing the identity of the

submission owner to illustrate good and bad programming habits.

For online courses or when the class time is limited, authors of this work also implemented a screen casting software, which allows recording narrated video screen captures and then made them available to students to review. However, keeping track of students' programming sessions and analyzing them can hardly be done using remote desktop or screen-cast applications. Because, content based information can not be extracted from sessions recorded by such applications. Moreover, these applications are not adequate for resource limited environments.

Finally in [7], instructors teach the programming process using a five-step, test driven, incremental process (STREAM). Every week there is a mandatory assignment. For lab examinations, they propose a method, where students are instructed to call upon a TA when they reach a checkpoint to show and demonstrate their solutions. Students approach to the development process as well as their solutions count in the final grade. To evaluate whether students really apply the suggested approach when no guidance is provided, they conduct an experiment. In this experiment, students solve assignments, while TAs observe and make note of any violations to the method taught. Authors report that all students followed the process they have been taught. It is unclear, whether students were aware of the aim of this experiment. If they were, it is quite possible that it would influence their programming behavior.

In summary, none of these methods provide a way for tracking students process while they work on their own. Thus, we see strong viability in favor of our tool in this context.

10. FUTURE WORK

Implementation of the Screen-Replay as a process tracking tool enabled us to replay students programming sessions and identify their programming activities by tagging them. We also implemented additional functionalities that support instructors in the tagging process. But, the tagging process is still time consuming. We are seeking ways to implement an extended version of our tool that supports automatic tagging. Such a tool will reduce the time required for tagging and eliminate possible mistakes during the tagging process.

Currently, Screen-Replay tool only records and replays text-based actions. DrScheme, however, enables students to insert images or other types of snips into the program text [8]. To be able to make more accurate analysis, our tracking tool will be enhanced with support for these types of snips.

Finally, we are planning to add support for recording the interaction window of DrScheme. This will allow the investigation for;

- When and how many times students run their programs?
- What are the common errors they get?
- Do students act according to the error messages?

which can not be answered just recording the definitions window.

11. CONCLUSION

Evaluating how students construct programs is difficult with conventional examinations as they evaluate the result and not the process. Evaluating student process requires observing how they construct their programs in a transparent manner. Else, we run the risk of altering their behavior.

We have developed a tool for transparently observing how students develop their programs. This tool was specifically designed to identify the sequence of activities in terms of the "How to Design Programs" (HtDP) methodology. The tool was implemented and integrated into the DrScheme environment. Screen-replay worked well, as it perfectly revealed the entire development process of students.

We also developed a process scoring algorithm to assess how well students follow design recipes. An experiment with 61 students is conducted and program development activities are recorded. Recorded process data is replayed, annotated and scored. Resulting scores are used together with the exam grades to examine the impact of using the design recipe.

We refrain drawing too strong conclusions from this experiment, but, the results indicate there is a relation between the application order of the design recipe and the grade acquired from the exam. Students, who had higher process scores, acquired higher grades from the exam. In other words, applying the design recipe in the suggested order resulted in higher grades.

APPENDIX A: Experiment Data: Process Scores and Exam Grades

Table A.1. Process scores and exam grades of students (1)

	Process scores			Exam grades		
Student Id	Main function	Helper functions	Exam	Main function	Helper functions	Exam
31	87	72	78	82.5	37.13	55.28
33	13	75	50.2	38	47	43.4
34	34	45	40.6	32.5	61.17	49.7
69	60	0	24	67.5	0	27
77	14	54	38	20	55	41
132	73	77	75.4	51	49	49.8
134	79	99	91	81.25	55	65.5
137	57	0	22.8	66.25	0	26.5
144	98	35	60.2	56	100	82.4
149	14	87	57.8	32	0	12.8
155	78	0	31.2	63	0	25.2
163	90	0	36	81	0	32.4
165	14	63	43.4	20	57.67	42.6
166	64	79	73	60.5	100	84.2
167	69	0	27.6	43	0	17.2
175	73	19	40.6	52	0	20.8
176	73	27	45.4	43	18	28
177	80	0	32	50.5	0	20.2
191	77	48	59.6	33	70	55.2

Table A.2. Process scores and exam grades of students (2)

	Process scores			Exam grades		
Student Id	Main function	Helper functions	Exam	Main function	Helper functions	Exam
192	14	37	27.8	20	21.33	20.8
193	67	19	38.2	77.5	22.5	44.5
199	89	71	78.2	68	34	47.6
207	24	48	38.4	20	72.63	51.58
208	82	0	32.8	53.5	0	21.4
214	83	0	33.2	40	0	16
215	54	26	37.2	39	55	48.6
216	51	66	60	68.5	84.38	78.03
219	48	58	54	14	25	20.6
220	74	76	75.2	77.5	73.06	74.84
221	71	78	75.2	46.5	56	52.2
222	74	36	51.2	67.75	55	60.1
226	94	0	37.6	28.5	0	11.4
234	77	65	69.8	83.75	32	52.7
238	87	0	34.8	61	0	24.4
241	78	93	87	50.75	78.75	67.55
244	85	93	89.8	41.75	43.5	42.8
245	73	68	70	46	36	40
249	56	81	71	82.5	91.56	87.94
250	73	87	81.4	57	84.75	73.65
252	93	0	37.2	61	0	24.4

Table A.3. Process scores and exam grades of students (3)

	Process scores			Exam grades		
Student Id	Main function	Helper functions	Exam	Main function	Helper functions	Exam
255	65	69	67.4	46	49	47.8
258	84	0	33.6	88.75	0	35.5
262	92	85	87.8	45.25	40	42.1
263	64	19	37	8	0	3.2
267	85	41	58.6	30	56.5	45.9
271	67	0	26.8	34	0	13.6
272	77	48	59.6	37.75	37	37.3
280	79	64	70	31.75	27.75	29.35
301	57	19	34.2	39	0	15.6
306	82	51	63.4	51	52	51.6
309	86	30	52.4	61	33	44.2
316	14	0	5.6	8	0	3.2
327	66	0	26.4	53	0	21.2
331	79	38	54.4	44.25	48.94	47.06
333	93	0	37.2	43	0	17.2
469	29	0	11.6	12	0	4.8
471	88	48	64	80.25	68.75	73.35
475	73	24	43.6	90	25.31	51.19
477	14	69	47	36	0	14.4
536	93	94	93.6	100	96.25	97.75
537	73	0	29.2	76.25	0	30.5

REFERENCES

1. Bieniusa, A., M. Degen, P. Heidegger, P. Thiemann, S. Wehr, M. Gasbichler, M. Sperber, M. Crestani, H. Klaeren and E. Knauel, "Htdp and dmda in the battlefield: a case study in first-year programming instruction", *FDPE '08: Proceedings of the 2008 international workshop on Functional and declarative programming in education*, pp. 1–12, ACM, New York, NY, USA, 2008.

2. Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi, "The structure and interpretation of the computer science curriculum", *J. Funct. Program.*, Vol. 14, No. 4, pp. 365–378, 2004.

3. Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi, "The TeachScheme! project: Computing and programming for every student", *Computer Science Education*, Vol. 14, No. 1, 2004.

4. Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi, *How To Design Programs*, MIT Press, Cambridge, MA, USA, 2001.

5. Klaeren, H. and M. Sperber, *Die Macht der Abstraktion*, Teubner Verlang, 1st edn., 2007.

6. Daly, C. and J. Waldron, "Assessing the assessment of programming ability", *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pp. 210–213, ACM, New York, NY, USA, 2004.

7. Bennedsen, J. and M. Caspersen, "Assessing Process and Product - A Practical Lab Exam for an Introductory Programming Course", pp. 16–21, Oct. 2006.

8. Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen, "DrScheme: a programming environment for Scheme", *J. Funct. Program.*, Vol. 12, No. 2, pp. 159–182, 2002.

9. Sperber, M., R. K. Dybvig, M. Flatt, A. Van Straaten, R. Finddler and J. Matthews, "Revised6 Report on the Algorithmic Language Scheme", *Journal of Functional Programming*, Vol. 19, No. Supplement S1, pp. 1–301, 2009, http://journals.cambridge.org/action/displayAbstract?fromPage=online&aid=6046168&fulltextType=RA&fileId=S0956796809990074.

10. Matthias Felleisen, "Matthias Felleisen", http://www.ccs.neu.edu/home/matthias/, 01 2010.

11. PLT-Scheme, "How to Design Programs Languages", http://docs.plt-scheme.org/htdp-langs/index.html, 02 2010.

12. Clements, J., M. Flatt and M. Felleisen, "Modeling an Algebraic Stepper", *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pp. 320–334, Springer-Verlag, London, UK, 2001.

13. LGPL, "GNU Lesser General Public License", http://cs.bilgi.edu.tr/pages/courses/year_1/comp_111/archive/2004-2005_Spring/robot_world/, 01 2010.

14. The TeachScheme! Project, "TeachScheme, ReachJava!", http://www.teach-scheme.org/, 01 2010.

15. Bloch, S., "Teach Scheme, reach Java: introducing object-oriented programming without drowning in syntax", *Journal of Computing Sciences in Colleges*, Vol. 23, No. 5, pp. 65–67, 2008.

16. Istanbul Bilgi University Department of Computer Science, "Bilgi University Index", http://www.cs.bilgi.edu.tr, 01 2010.

17. Istanbul Bilgi University Department of Computer Science, "Bilgi University Courses Year 1 Comp 149 Index", http://www.cs.bilgi.edu.tr/pages/courses/year_1/comp_149, 01 2010.

18. Stephenson, C., "RobotWorld Home Page", http://cs.bilgi.edu.tr/pages/courses/year_1/comp_111/archive/2004-2005_Spring/robot_world/, 01 2010.

19. Pattis, R. E., *Karel the Robot: A Gentle Introduction to the Art of Programming*, John Wiley & Sons, Inc., New York, NY, USA, 1981.

20. Bergin, J., J. Roberts, R. Pattis and M. Stehlik, *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*, John Wiley & Sons, Inc., New York, NY, USA, 1996.

21. Alphonce, C. and P. Ventura, "Using graphics to support the teaching of fundamental object-oriented principles in CS1", *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 156–161, ACM, New York, NY, USA, 2003.

22. Cooper, S., W. Dann and R. Pausch, "Alice: a 3-D tool for introductory programming concepts", *CCSC '00: Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges*, pp. 107–116, Consortium for Computing Sciences in Colleges, , USA, 2000.

23. Sanders, D. and B. Dorn, "Jeroo: a tool for introducing object-oriented programming", *SIGCSE Bull.*, Vol. 35, No. 1, pp. 201–204, 2003.

24. PLT-Scheme, "PLT Scheme", http://www.plt-scheme.org/, 01 2010.

25. Polya, G., *How to Solve It (Penguin Science)*, Penguin Books Ltd, April 1990, http://www.amazon.de/exec/obidos/redirect?tag=citeulike01-21\&path=ASIN/0140124993.

26. Proulx, V. K. and T. Cashorali, "Calculator problem and the design recipe", *SIGPLAN Not.*, Vol. 40, No. 3, pp. 4–11, 2005.

27. Proulx, V. K. and K. E. Gray, "Design of class hierarchies: an introduction to

OO program design", *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pp. 288–292, ACM, New York, NY, USA, 2006.

28. Bloch, S. A., "Scheme and Java in the first year", *J. Comput. Small Coll.*, Vol. 15, No. 5, pp. 157–165, 2000.

29. Köksal, M. F., R. E. Başar and S. Üsküdarlı, "Screen-Replay: A Session Recording and Analysis Tool for DrScheme", *Proceedings of the 2009 Scheme and Functional Programming Workshop*, pp. 103–110, 2009.

30. PLT-Scheme Reference, "5.2 Creating classes", `http://docs.plt-scheme.org/reference/createclass.html?q=definitions-text&q=%s`, 02 2010.

31. Navarro, G., "A guided tour to approximate string matching", *ACM Comput. Surv.*, Vol. 33, No. 1, pp. 31–88, 2001.

32. Levenshtein, V., "Binary codes capable of correcting spurious insertions and deletions of ones", *Problems of Information Transmission*, Vol. 1, pp. 8–17, 1965.

33. The Levenshtein-Algorithm, "Efficient Implementation of the Levenshtein-Algorithm, Fault-tolerant Search Technology, Error-tolerant Search Technologies", `http://www.levenshtein.net/`, 01 2010.

34. Wikipedia, "Levenshtein distance", `http://en.wikipedia.org/wiki/Levenshtein_distance`, 01 2010.

35. Istanbul Bilgi University, "Academic Regulations", `http://www.bilgi.edu.tr/pages/statics.asp?mmi=12&stbl=sub9&id=12&sid=13&r=04.02.2010+11%3A37%3A41`, 02 2010.

36. Box, G. E. P., J. S. Hunter and W. G. Hunter, *Statistics for Experimenters: Design, Innovation, and Discovery (2nd)*, John Willey and Sons Inc., 2005.

37. Saff, D. and M. D. Ernst, "An experimental evaluation of continuous testing during development", *SIGSOFT Softw. Eng. Notes*, Vol. 29, No. 4, pp. 76–85, 2004.

38. Gaspar, A. and S. Langevin, "Restoring "coding with intention" in introductory programming courses", *SIGITE '07: Proceedings of the 8th ACM SIGITE conference on Information technology education*, pp. 91–98, ACM, New York, NY, USA, 2007.