Screen-Replay: A Session Recording and Analysis Tool for DrScheme

M. Fatih Köksal, R. Emre Başar

Department of Computer Science
İstanbul Bilgi University
{fkoksal,reb}@cs.bilgi.edu.tr

Suzan Üsküdarlı

Department of Computer Engineering
Boğaziçi University
suzan.uskudarli@boun.edu.tr

Abstract

Approaches to teaching "Introduction to Programming" vary considerably. However, two broad categories may be considered: product oriented vs process oriented. Whereas, in the former the final product is most significant, in the latter the process for achieving the final product is also considered very important. Process oriented programming courses strive to equip students with good programming habits. In such courses, assessment is challenging, since it requires the observation of how students develop their programs. Conventional methods and tools that assess final products are not adequate for such observation.

This paper introduces a tool for non-intrusive observation of program development process. This tool is designed to support the process oriented approach of "How to Design Programs" (HtDP) and is implemented for the DrScheme environment. The design, implementation and utility of this tool is described with examples.

Keywords Introductory Programming, Development Process, Design Recipe, DrScheme

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 [2]. With these tools and course instruction students are expected to learn how to write, debug and document programs.

While the objectives of introductory programming courses are similar, the content, approach and assessment methods differ. Teaching with examples is frequently used [6], 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 result. The students code in this case is the final product. There is no further information on how the student arrived at the final product.

The TeachScheme! project [7] does not appreciate the programming-by-tinkering methodology. It developed an alternative approach to teaching, described in the text book, "How to Design Programs" (HtDP) [5]. This approach focuses on a design process that starts from problem statement to a well-organized solution. After the publication of HtDP, several universities around the world revised their curriculum in favor of this approach. Most universities use the methodology as described in the book, where others [2] have derived versions [10] according to their needs.

The HtDP and approaches derived from it emphasize the importance of process in comparison 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 students progress [4]. Approaches to conduct live exams also vary. Some let students develop programs independently and evaluate results in a conventional manner. In others [2, 1] the development process is observed personally. The observation process is an intrusive approach that may impact student performance.

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 following questions: Do students follow the suggested design guidelines while they develop programs on their own? Are students, who follow the suggested guidelines, more successful than the others? If not, is there any specific design pattern that is commonly used by successful students? Using an intrusive tracking method may impact students' performance in the programming session. Indeed, it has been reported that some students were disturbed by personal observation of their work [1].

An alternative approach for tracking program development is to embed the tracking ability into the development tool. Such a tool would need to record as well as replay the development process. This work describes a program development tracking tool for DrScheme [8] that enables a student to record his/her programming session. This recorded session can, then, be replayed and analyzed by an observer.

The rest of this paper is organized as follows: Section 2 further discusses our motivation to analyze students' programming sessions in order to answer questions we stated above. Section 3 investigates related work regarding product and process oriented approaches and their assessment techniques. Underlying concepts and implementation details are given in Section 4, followed by a discussion in Section 5. Finally, in Sections 6 and 7, we discuss future work to be done and conclude our work.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

2. Motivation

The first year curriculum for Computer Science Department at İstanbul Bilgi University was revised effective of 2007-2008 academic year. Courses were divided into sections of at most 20 students, in order to have better control over the course and increase student-instructor interaction. With this change, we have been able to intensively follow our students to see if they meet our educational approaches.

The 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 to Computer Science, Financial Mathematics and Business Informatics majors. Comp149/150-HtDP uses "How to Design Programs" (HtDP) [5] 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 write 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 complicated and students are given at least two weeks to complete. During lab sessions students present their project solutions to their classmates.

During this course students are given four live exams. Each exam consists of one or two questions that have to be solved in approximately 1.5 hours. Exams are completed on computers, where students only have access to the text book and DrScheme. All networking is disabled during the exams. Grades of weekly projects and live exams determine the course grade of students. Final grade of a student from this course is combined with grades from other parts of the meta-course using 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, students are examined by a jury at the end of the year by their instructors of this meta-course.

The main objective of the entire course is to teach "How to solve it?" [11] and the process is central to this idea. The following section describes the design recipe methodology of HtDP that, in theory, meets the aim of our introductory programming course.

2.1 HtDP and the Design Recipe

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 [5]. 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 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. Basic steps of the design recipe are as follows;

0. *Data definition*: describe the class of problem data
1. *Contract*: name your function and give input-output relation in terms of data type used
2. *Purpose*: informally specify the behavior of your program
3. *Examples*: illustrate the behavior with examples
4. *Template*: develop your programs template/layout
5. *Code*: transform your template into a complete definition
6. *Tests*: turn your examples into formal test cases.

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

Students are expected to use this checklist on a question-and-answer basis to progress towards a solution [5]. Figure 1 shows the application of a design recipe for summing the elements of a list.

2.2 The Strategic War Between Instructors and Students

There are numerous reports of success using HtDP curriculum [12, 2, 13, 3]. Since the adoption of HtDP, we have also observed similar improvements. Specifically, we have observed improvements in student performance with respect to:

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

These improvements are particularly noticeable in female students.

On the other hand, increased interaction with students revealed some deficiencies in their adoption of the process we use. Students were not applying the design recipe throughout their development process. They were diving into the code without going through the design steps. To tackle this problem, a change in our grading scheme was required. The grading scheme was changed to grade every step of the design recipe separately.

Students responded by faking the process. They were writing the code first and adding the design steps later. This response led us to inspect each student submission more carefully. The forged design steps can be distinguished by checking the inconsistencies between the steps. Considering that, our response was to do a consistency check between the design steps and stopping evaluation of the assignment when an inconsistency was found.

At that point, it was understood that applying more force on following the recipe only created better "design recipe evasion" tactics. With this realization we abandoned the attempt to evaluate the order of construction and only verified presence of correct parts. Currently, the recipe is followed while teaching, and students are encouraged to use for every program they develop. But, the application of the recipe is not enforced or evaluated in any way.

However, we are still interested in tracking our students' development processes to see both how they develop their programs and whether the suggested approach helps them to build well-structured solutions. Therefore we developed a tool for just that purpose.

3. Related Work

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

In [14], authors report on a controlled experiment to evaluate whether students using continuous testing are more success-

```scheme
;; Data definition:
;; 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:
;; sum-lon: lon -> number

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

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

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

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

;; Tests:
(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 1. Application of the design recipe for summing the elements of a list

ful in completing programming assignments. As the source code is edited, continuous testing uses excess cycles on a developer's workstation to continuously run regression tests in the background against the current version of the code providing feedback about test failures. Their tool aim to give extra feedback during the programming session and improve the productivity of developers. The experimental results indicate that students using continuous testing 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 [2], 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 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. In our experiences, on the other hand, 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 authors state, nearly 15% 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.

This 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 proactively 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 to 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 [9] 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. Interfering as they make mistakes means taking their chance of solving the problem by themselves, therefore learning the importance of design recipe.

Exposing a student's errors 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 using in online courses or when the class time is limited, authors of this paper also implemented a screen casting software which allows to record narrated video screen captures and then made available to students to review. Keeping track of students' programming sessions and analyzing them can hardly be done using remote desktop or screen-cast applications. 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 [1], instructors teach the programming process using a five steps, 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,

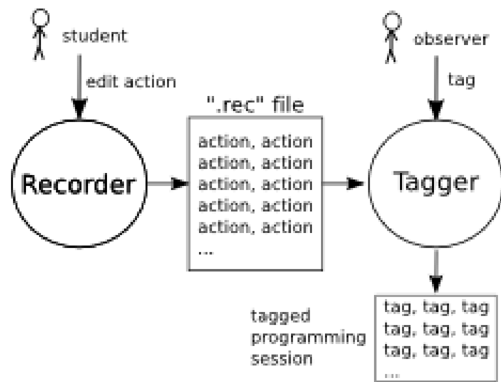

Figure 2. An overview of the Screen-Replay tool

they conduct an experiment. In this experiment students solve the 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 inpact their programming behaviour.

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.

4. Tracking the Development Process

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

4.1 Requirements

The fundamental requirements of the system are:

1. Record every state the program takes during its construction lifetime. The lifetime begins with the creation of the program session until its completion.
2. Replay the construction of the program.
3. Describe the high level programming activities taken during the program constrution. These activities are the ones described by the HtDP methodology.

The requirement 1 must be satisfied within the development environment in a transparent manner. In another words, construction activities must be recorded in the background while the student is constructing their solution. Requirements 2&3 are meant for evaluators who will inspect and annotate the students program construction.

4.2 Implementation

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

4.2.1 Recorder

The Recorder records all user interactions within the DrScheme's *Definitions* window, which is where programs are defined. The Recorder saves information about any insertion or deletion. The following Scheme structure, *action*, describes the information stored for every user interaction.

```
;; 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)
```

The following example is an action that indicates that user typed *f*, an insertion of length 1, at position 0 of the definitions window. Position 0 is the starting position.

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

For every text insertion and deletion the Recorder creates a corresponding action. Actions remain in the buffer until the file is saved. The Recorder catches keystrokes by extending `definitions-text` with a mixin. This mixin augments the `insert` and `on-delete` methods with use of a boolean flag to indicate the recording state of the current window. This approach makes it possible to record actions in each window separately.

When a file is saved the buffer content is written to an *actions-file* with a ".rec" extension. An actions-file consists of a series of action structures serialized with the `write` function. 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. Recorded files are replayed using the Tagger.

4.2.2 Tagger

The Tagger has two main functions: (1) To replay the program construction and (2) describe the high-level construction process in terms of the HtDP methodology.

The Tagger allows the observer to see exactly how the program was constructed. While observing the construction process, the observer can describe the programming activity using tags defined for this purpose.

Replaying: The Tagger replays the exact steps taken while the program was written. 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

A time slider is supplied to enable the observer to 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 Tagger automatically scrolls to the position within the program that is associated

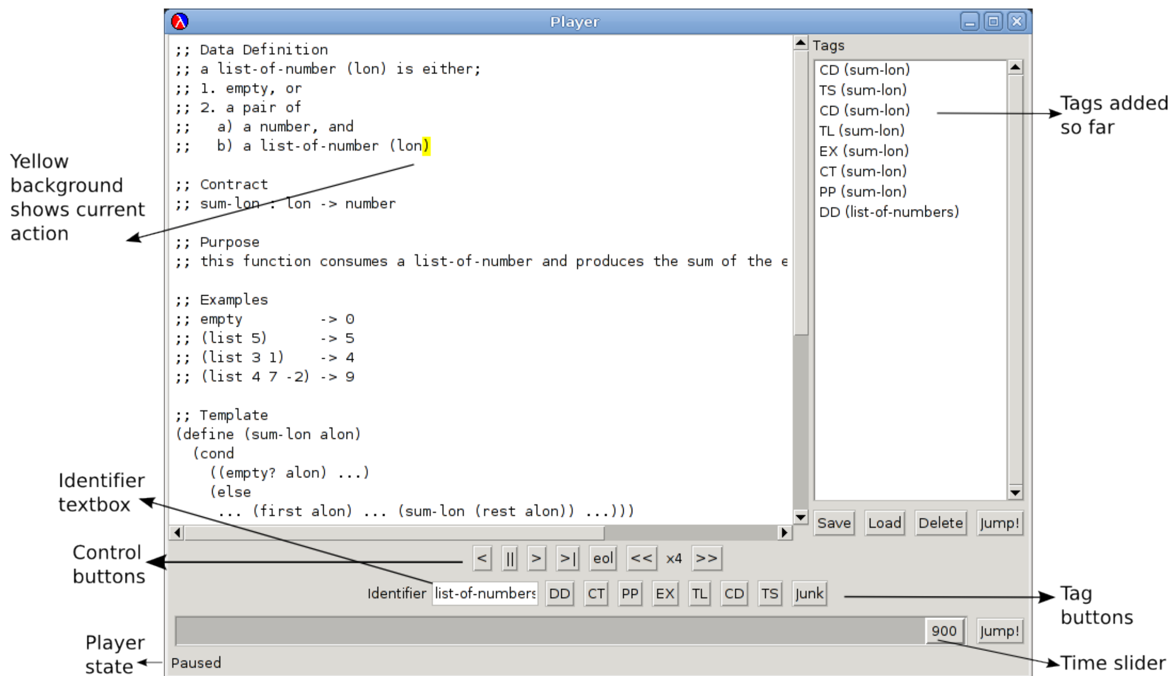

Figure 3. An overview of the Tagger tool

with the current action. This makes the location visible making it easier to follow the flow of construction. The currect action record is shown with a yellow highlight.

When a file is selected to be played, all actions in the actions-file are loaded into a *Tape* structure defined as follows:

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

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

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

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

will insert the single character "f" in the editors first position. 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 Tagger replays the construction at the same speed of the original construction. Scheme semaphores are used in order to make the Tagger 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 the recorded example shown in Figure 1. The final product shows that all design recipe steps are present. The Tagger enables one to view the process that led to this product. The first column of Figure 4 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 fool the instructor. The student first implemented the code and then added the remaining required steps. The tracking process reveals the order of the application of design recipe. It also shows how much time is spent on each step. The ability to observe such a process enables the instructor to discover deficiencies and provide more accurate help.

Tagging: Tagging allows the evaluator to describe that kinds of activities performed during the constructing of a program. Recall that the activities of interest in HtDP are:

0. Data definition
1. Contract
2. Purpose
3. Examples
4. Template
5. Code
6. Tests

To identify each of these activities the corresponding tags DD, CT, PP, EX, TL, CD and TS are defined. During the tagging process the observer specifies a sequence of tags as he/she observes the construction states. The interface includes buttons for each tag. The observer clicks, for example, to the DD button, when the student finishes editing the data definition and moves to another state.

It is possible that the student performs some activity that does not fit within the HtDP methodology. For this, a Junk tag was defined. 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 enumerated tags defined earlier.

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.

Tags are stored in *Tag* structure:

```
;; tag
;; name (symbol)        : name of the tag
;; identifier (symbol): an identifier that
;;                        the tag is tied to
;; end (number)         : value of the tape-pointer
;;                        at the time of tagging

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

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

The development process is denoted with a sequence of tags, which are inserted by clicking on the appropriate tag-button. Furthermore, an identifier can be associated with each tag to further describe which function the activity is associated to. 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 programming assignment in Figure 1, which we assumed to be recorded using the Recorder. Second column of the Figure 4 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 example below, is generated. Tag-end positions may not be easily traceable from the given figure, 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))
```

Above tag-list, generated from the tagging session, tells us that actions from indices 0 up to 158 are somehow related and have the same context (they form the code for sum-lon function for this particular example). Similarly, actions between 159-297, 298-303 (and so on) have their own context according to the observer. Therefore he/she generated new tags.

The application of design recipe was already revealed with 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.

Tag-list, by itself, includes some information about the session and can be used for investigation 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.

4.3 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 already available time and position data in actions, tags can be extended with more information to generate a self contained analysis data. Processed-tag is defined as follows to meet this requirement;

```
;; 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)
```

As described above, processed-tag includes much more information about the actions associated 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 progress 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.

5. Discussion

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[1], since this feature has not yet been implemented.

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 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.

The fuzzy nature of the design recipe makes it hard to automatically detect the design segments. Students apply it in different

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

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 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 the examples according to tests.	Pushes CD, then TL as the student finishes writing code and template, respectively. Waits the student 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. Realizes that the student is writing data definition for the list-of-number. Updates the identifier. 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.

Figure 4. Students actions and observers responses in return

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-lines jump occur.

6. Future Work

Implementation of the Screen-Replay as a process tracking tool enabled us to investigate the efficiency of the teaching methods we use. Last three live exams (approximately 100 students each) of our introductory programming course are already recorded. After the end of tagging process we will investigate answers to the following questions;

- Do students follow the suggested design guidelines while they develop programs on their own?
- Are students, who follow the suggested guidelines, more successful than the others?
- If not, is there any specific design pattern that is commonly used by successful students?

Currently, our tool only records and replays text-based actions. To be able to make more accurate analysis, the Screen-Replay tool will be enhanced with support for images or other 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.

7. 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 was used to record over 100 program consructions during live examinations. Replaying these constructions enabled observers to see the exact manner in which the programs were constructed. With the Tagger tool observers were able to associate student activity with a segment of the construction. Finally, a program construction is associated with a sequence of tags that describes the entire constrution process.

Screen-replay worked well, as it perfectly revealed the entire development process of students. We find the observation process to be very interesting and insightful. It is too early to make any conclusions as of yet. Screen-replay is being used to analyze results of student examinations. The results of the analysis will be reported. Improvements to the tool are also in progress, especially in terms of improving the tagging process.

Acknowledgments

We would like to thank Vehbi Sinan Tunalıoğlu and Bülent Özel for their support to improve this tool and paper. The work in this paper is partially funded by the Boğaziçi University Research Fund BAP 07A107 and 08A103.

References

[1] J. Bennedsen and M.E. Caspersen. Assessing process and product - a practical lab exam for an introductory programming course. pages 16–21, Oct. 2006.

[2] Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Michael Sperber, Marcus Crestani, Herbert Klaeren, and Eric Knauel. Htdp and dmda in the battlefield: a case study in first-year programming instruction. In *FDPE '08: Proceedings of the 2008 international workshop on Functional and declarative programming in education*, pages 1–12, New York, NY, USA, 2008. ACM.

[3] Stephen A. Bloch. Scheme and java in the first year. *J. Comput. Small Coll.*, 15(5):157–165, 2000.

[4] Charlie Daly and John Waldron. Assessing the assessment of programming ability. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 210–213, New York, NY, USA, 2004. ACM.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How To Design Programs*. MIT Press, Cambridge, MA, USA, 2001.

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *J. Funct. Program.*, 14(4):365–378, 2004.

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The teachscheme! project: Computing and programming for every student. *Computer Science Education*, 14(1), 2004.

[8] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

[9] Alessio Gaspar and Sarah Langevin. Restoring ”coding with intention” in introductory programming courses. In *SIGITE '07: Proceedings of the 8th ACM SIGITE conference on Information technology education*, pages 91–98, New York, NY, USA, 2007. ACM.

[10] Herbert Klaeren and Michael Sperber. *Die Macht der Abstraktion*. Teubner Verlang, 1st edition, 2007.

[11] George Polya. *How to Solve It (Penguin Science)*. Penguin Books Ltd, April 1990.

[12] Viera K. Proulx and Tanya Cashorali. Calculator problem and the design recipe. *SIGPLAN Not.*, 40(3):4–11, 2005.

[13] Viera K. Proulx and Kathryn E. Gray. Design of class hierarchies: an introduction to oo program design. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 288–292, New York, NY, USA, 2006. ACM.

[14] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. *SIGSOFT Softw. Eng. Notes*, 29(4):76–85, 2004.