Monitoring Interface and Automated Testing for Seaweed, a Web-based Economic Game System

by

Jessica Yuan

S.B., EECS, M.I.T., 2009

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2010

[June 2010]

Author____________________________________
Department of Electrical Engineering and Computer Science
May 17, 2010

Certified by______________________________
Professor [illegible]
Thesis Supervisor

Accepted by______________________________
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Monitoring Interface and Automated Testing for Seaweed, a Web-based Economic Game System

by

Jessica Yuan

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 2010

In partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Seaweed is a web-based economic game system that allows end users to design and deploy simple two-player economic games. To improve the usability of the system, we have created two new features. One: a monitoring interface that allows users to view the results of a game deployment in real time. Two: an automated testing feature that automatically generates play data for a game, so that the user can check that the results are correct. Challenges included condensing the data of multiple plays of a game into a readable format and modifying the computer player system to pit two computer players against each other in order to generate the test data.

Thesis Supervisor: Robert C. Miller
Title: Associate Professor

Acknowledgments

I would like to thank Lydia Chilton, who worked very hard to create Seaweed and was kind enough to allow me to contribute to her project. I will never forget her brilliance and enthusiasm.

I would also like to thank Joey Lawrance for his help in working on this project. He was always willing to give a hand when I was stuck, and he spent numerous hours helping me get the Seaweed code working on my machines.

My greatest thanks go to my advisor Rob Miller and his invaluable experience, patience, and support. Because of him, I have learned much about web programming and software and UI design.

Thanks also to my parents, who have supported me all along the way. I would also like to thank the wonderful friends I have made here at MIT. Although we may part ways soon, I will always cherish the times we had.

Contents

List of Figures

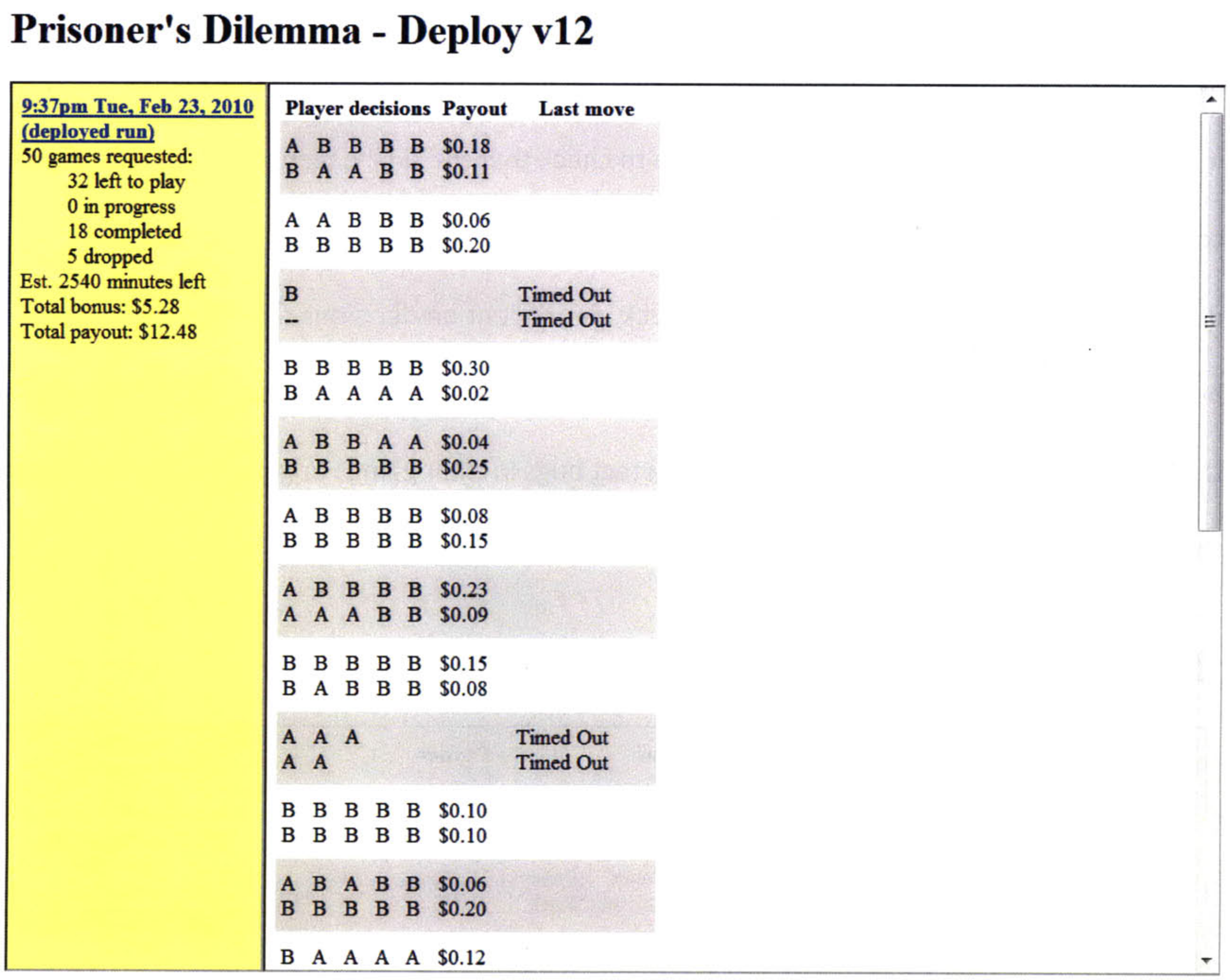
1. Introduction

Seaweed is a web application that allows users to design, deploy, and play economic games. Its target audience is experimental economists, who may not have the time or programming experience necessary to program their own game system. Seaweed is an *end user programming* platform, allowing users who are not professional programmers or developers to design and implement online economic games with relative ease [10].

In their research, experimental economists design new games and have research subjects play them. Generally, the economists must hire programmers to write the game software, and then they must enlist participants to come into the computer lab to play the game, usually by offering some sizable monetary incentive. This process can be laborious and expensive, especially if the economists decide to make changes to their game. Seaweed reduces the time and cost necessary for economists to design and deploy their games. Its end-user designer interface allows the economists to program and make changes to their games quickly and easily, and Seaweed's deployment mechanism lets economists publish the game on the web, where they have access to thousands of potential participants.

To improve the usability of Seaweed, for this thesis, we added two features:

1. A monitoring interface where economists can view the status and results of ongoing games in real time
2. An automated tester which allows economists to run sample games and examine if the games are behaving as expected


Prisoner's Dilemma - Deploy v12

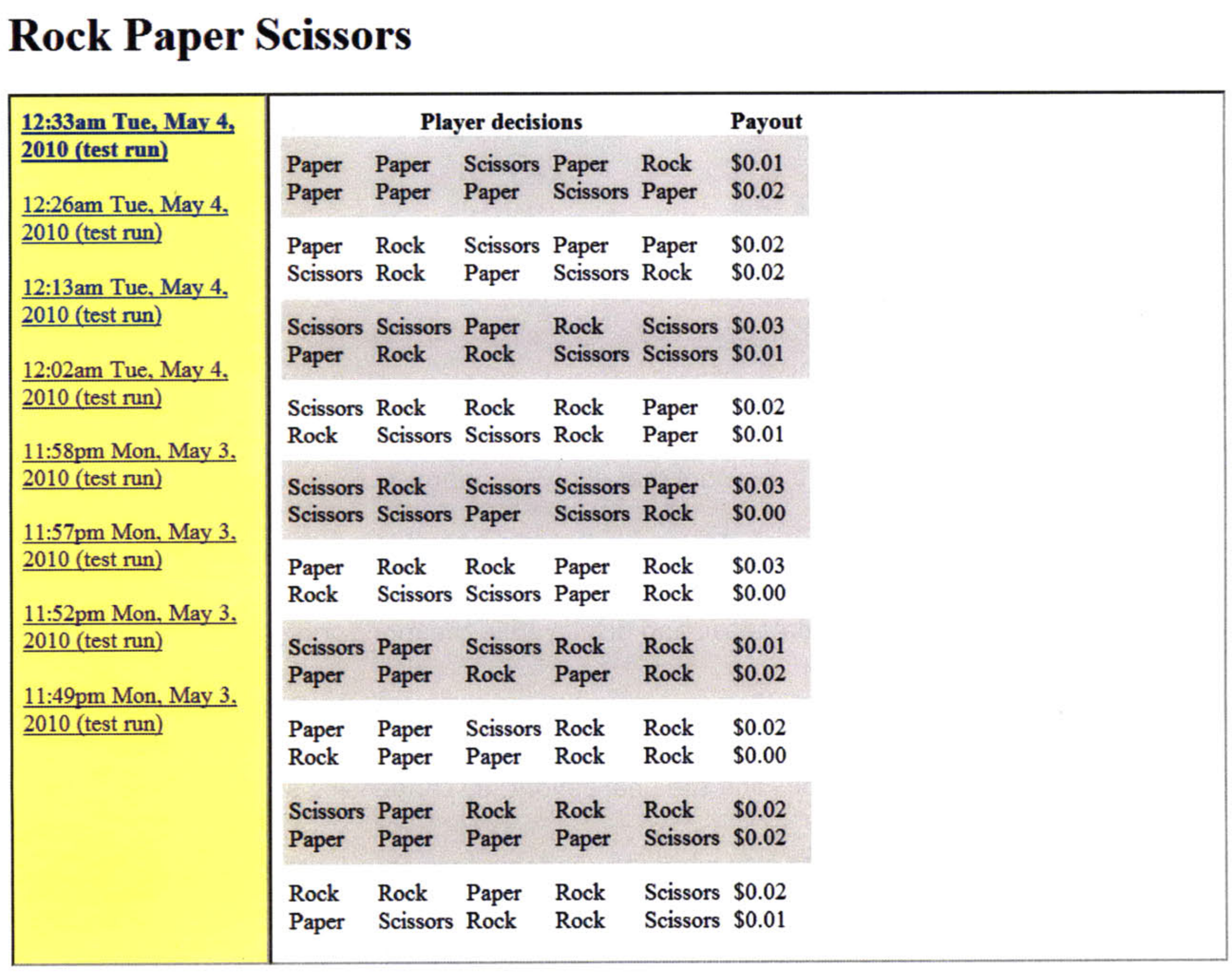
9:37pm Tue, Feb 23, 2010
(deployed run)
50 games requested:
32 left to play
0 in progress
18 completed
5 dropped
Est. 2540 minutes left
Total bonus: $5.28
Total payout: $12.48

Player decisions	Payout	Last move
A B B B B	$0.18	
B A A B B	$0.11	
A A B B B	$0.06	
B B B B B	$0.20	
B		Timed Out
--		Timed Out
B B B B B	$0.30	
B A A A A	$0.02	
A B B A A	$0.04	
B B B B B	$0.25	
A B B B B	$0.08	
B B B B B	$0.15	
A B B B B	$0.23	
A A A B B	$0.09	
B B B B B	$0.15	
B A B B B	$0.08	
A A A		Timed Out
A A		Timed Out
B B B B B	$0.10	
B B B B B	$0.10	
A B A B B	$0.06	
B B B B B	$0.20	
B A A A A	$0.12	

Figure 1. The monitoring interface.

The monitoring interface (Figure 1) is a tool that allows economists to observe the progress of ongoing games. As players join the game, get partnered up, and begin making decisions, those actions are written to the server database and displayed in the monitoring interface. The left sidebar provides a summary of all the games, and the right panel shows the details of each game, such as the players' decisions and the final payout. By condensing an otherwise overwhelming amount of play data into a human-readable format in real time, the monitoring interface gives economists a valuable window into how their games are progressing.

The automated tester is a feature that automatically plays a Seaweed game, selecting random decisions for both players and generating play data. It then displays this data using the monitoring interface. This allows economists to check that the final payouts are being computed properly. The automated tester can play a game multiple times, and it can also detect deadlocks, where both players are stuck, dependent on decisions that could not have been made yet. By providing an easy way to test numerous decision paths and detect deadlocks, the automated tester can help economists detect bugs in their games or gain confidence about their games' correctness.



Rock Paper Scissors

12:33am Tue, May 4, 2010 (test run)

12:26am Tue, May 4, 2010 (test run)

12:13am Tue, May 4, 2010 (test run)

12:02am Tue, May 4, 2010 (test run)

11:58pm Mon, May 3, 2010 (test run)

11:57pm Mon, May 3, 2010 (test run)

11:52pm Mon, May 3, 2010 (test run)

11:49pm Mon, May 3, 2010 (test run)

Player decisions					Payout
Paper	Paper	Scissors	Paper	Rock	$0.01
Paper	Paper	Paper	Scissors	Paper	$0.02
Paper	Rock	Scissors	Paper	Paper	$0.02
Scissors	Rock	Paper	Scissors	Rock	$0.02
Scissors	Scissors	Paper	Rock	Scissors	$0.03
Paper	Rock	Rock	Scissors	Scissors	$0.01
Scissors	Rock	Rock	Rock	Paper	$0.02
Rock	Scissors	Scissors	Rock	Paper	$0.01
Scissors	Rock	Scissors	Scissors	Paper	$0.03
Scissors	Scissors	Paper	Scissors	Rock	$0.00
Paper	Rock	Rock	Paper	Rock	$0.03
Rock	Scissors	Scissors	Paper	Rock	$0.00
Scissors	Paper	Scissors	Rock	Rock	$0.01
Paper	Paper	Rock	Paper	Rock	$0.02
Paper	Paper	Scissors	Rock	Rock	$0.02
Rock	Paper	Paper	Rock	Rock	$0.00
Scissors	Paper	Rock	Rock	Rock	$0.02
Paper	Paper	Paper	Paper	Scissors	$0.02
Rock	Rock	Paper	Rock	Scissors	$0.02
Paper	Scissors	Rock	Rock	Scissors	$0.01

Figure 2. Monitoring interface for viewing automatically-generated test data.

The remainder of this thesis is structured in the following manner. Chapter 2 discusses related work. Chapter 3 describes Seaweed and how it is used. Chapter 4 covers the monitoring interface in detail, both design and implementation. Chapter 5 covers the automated testing feature. Chapter 6 is the conclusion. Chapter 7 discusses future work and the future of Seaweed.

2. Related Work

Automated testing is the use of computer programs to test software, as opposed to manual testing, which requires human testers. In his paper "When Should a Test Be Automated?" Brian Marick weighs the pros and cons of test automation [24]. Marick first describes several possible scenarios: that you already have access to test automation tools, that you must choose to either test entirely automatically or entirely manually, that the nature of what you are testing requires that you test automatically or manually, that you have the leeway of choosing what to test automatically and what to test manually, that you are limited in the amount of time given for testing, and some other scenarios.

Seaweed's users, being end user programmers, are unlikely to carefully consider which scenario is most closely related to their case. They are likely to be concerned with time and may not wish to spend a lot of time testing. With the automated testing feature, we give Seaweed's users access to a test automation tool. Seaweed also has a manual testing feature, Play Side by Side, which allows users to test parts of their game that cannot be tested automatically, such as layout and readability.

To answer the question of when a test should be automated, Marick makes his decision upon the answers to three questions:

1. Writing an automated test will cost more than doing manual testing once. How much more will it cost?

2. In the development process, an automated test will eventually become useless and need to be rewritten to handle an evolving code base. How long will the test automation be useful?
3. While the test is still alive and useful, how likely is it to uncover additional bugs, i.e. not including the bugs found on the first run of the automated test?

For typical Seaweed users, having to write a customized automated test for their game will cost a great deal without some sort of provided test framework, because of the users' inexperience with programming and testing principles. The customized automated test may not have a long lifetime if the user decides to modify the game. While the test is alive, it is unlikely to uncover any additional bugs; once the entire test has been passed, it will most likely continue to pass, as long as there are no major modifications to the game. Thus, it does not make much sense to ask the Seaweed user to generate an automated test.

Marick measures cost as the number of bugs potentially left unfound. Creating an automated test and running it costs more than running a manual test because the time it takes to create that automated test takes away from time spent doing manual testing and possibly finding new bugs. Although this idea of cost is unconventional, Marick argues that because the purpose of automated tests is to find new bugs on subsequent runs of the tests, the cost should also be measured in terms of new bugs. It's difficult to predict a quantitative number of potential bugs, so Marick claims that fuzzy values such as "probably none" are sufficiently useful when considering whether or not automated tests are worthwhile, especially when the alternative is making the same decision with less thought and information.

Andrew Ko et al. did a survey paper of the current research in end-user programming [19]. They define end-user programmers by their goals: professional programmers develop and maintain software for clients, but end-user programmers write programs to aid themselves. Statistics from the U.S. Bureau of Labor and Statistics report that although there will be fewer than 3 million professional programmers in the United States by 2012, there will be more than 55 million people using databases and spreadsheets for their work, many of whom will write queries and formulas to help themselves [35]. This provides the motivation for the survey, that end-user programming is becoming a prevalent trend.

In the context of the paper, they define a program to be "*a collection of specifications that may take variable inputs, and that can be executed (or interpreted) by a device with computational capabilities*." An end user is defined to be just any computer user, and end-user programming is "*programming to achieve the result of a program primarily for personal, rather public use*." By this definition, even professional developers are doing end-user programming when they are programming for themselves. The authors acknowledge that many researchers use the term end-user programming to mean novice programming or non-professional programming. In fact, in the context of Seaweed, we use the term end-user programming to denote programming done by users who may have little or no previous programming experience.

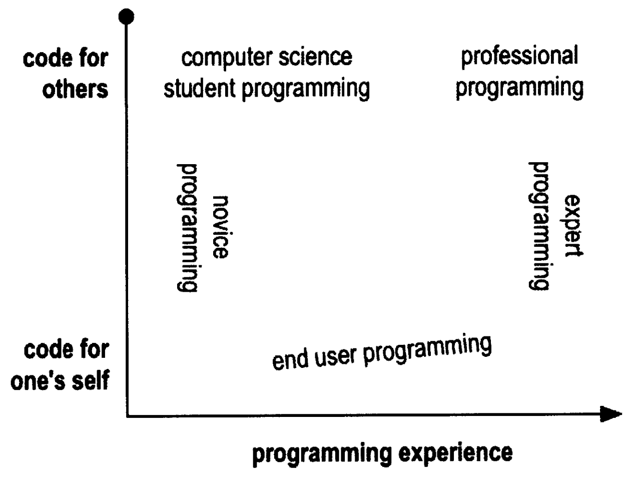


Figure 3. Ko et al. define end-user programming by intent, not experience.

Although end-user programmers have different priorities and motivations than professional developers, they encounter many of the same difficulties, including testing and debugging, and failures and errors may lead to financial or reputational damages. Ko et al. posit that the difference between end-user software engineering and professional software engineering is the amount of attention placed upon software quality concerns. Professional software engineering puts an extremely high emphasis on software quality because of the large client base it must satisfy. On the other hand, end-user software engineering is more concerned with achieving the end user's goals than having high software quality. Thus, the challenge of end-user software engineering is figuring out how to incorporate good software engineering practices into users' existing work style without significantly disrupting their work style or their priorities.

Ko et al. discuss five major aspects of end-user software engineering:

1. *Requirements.* What the software needs to do.
2. *Design and specifications.* How the software should be internally structured to satisfy the requirements.

3. *Reuse.* Saving time and preventing errors by using preexisting code.
4. *Testing and verification.* Discovering failures and achieving confidence about the software's correctness.
5. *Debugging.* Reducing failures by finding and correcting errors.

Requirements. While professional software engineers spend a great deal of time planning and specifying the requirements of their software, end-user programmers are not likely to do so. In fact, because end-user programmers are programming for themselves, they have a much better understanding of the requirements, and it's easier and more likely for the requirements to change. End users do not necessarily have to articulate these requirements if they have an intuitive understanding of what they need.

Design and specifications. Professional software designers take their requirements and translate them into design specifications, which describe the internal behavior of a system. End users may find this step difficult. Also, they may not be able to fully appreciate the advantages of explicit, well-designed specifications, which are most evident after long-term use. End users may expect that their programs will have short life spans, even if that turns out not to be the case. There has been research into systems that enforce consistent specifications [11] [34] or infer specifications from examples [21][1][25].

Reuse. Professional software engineers often reuse code, whether it's copying code, modifying code, or importing libraries or packages [6]. By doing so, they save time and eliminate the risk of introducing new buggy code [39][29]. The challenges facing end users include finding the appropriate code to reuse and figuring out how to use it [39]. There have been many tools developed that help programmers find useful abstractions or API functions [39][23][5][36][8]. Even after identifying a useful API, end users may struggle to use it properly and adapt it to meet their needs. Some platforms provide templates that the users can modify to suit their requirements [22].

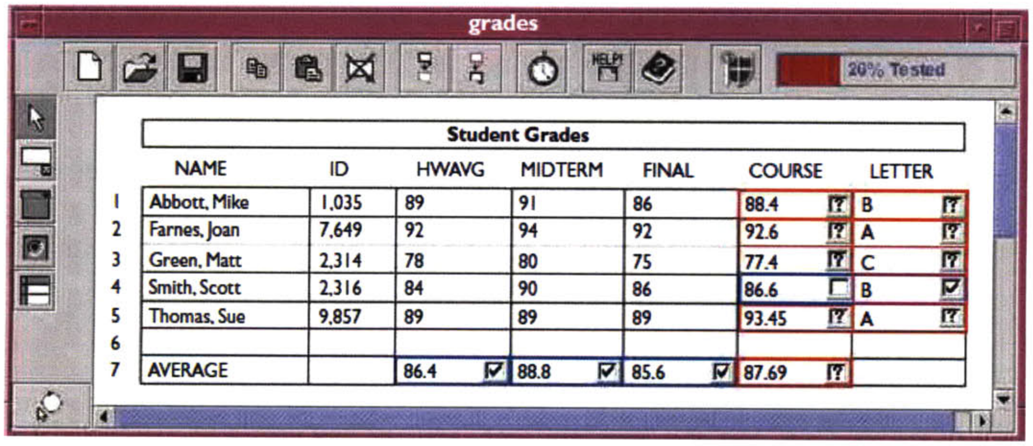
Another concern is that while end users may consider their program to be personal and short-lived, oftentimes their code is shared and reused. Thus, maintainability becomes an issue.

Testing and verification. The purpose of testing and verification is to gain confidence that a program is functioning correctly. End users are often overconfident about the correctness of their programs. In studies, they are more likely to indicate an incorrect value to be right than a correct value to be wrong [26][27][13][37][33][32][28].

One way to help end-user programmers find bugs is systematic testing, testing with a plan for what to test and how much to test. However, end users may be reluctant to do systematic testing, since it is time-consuming and not immediately relevant to their goals. Because of this, researchers have been focusing on developing testing tools that are integrated with end users' work.

Margaret Burnett et al. have researched testing tools for spreadsheets [9]. They use a methodology called What You See Is What You Test (WYSIWYT), where spreadsheet users are encouraged to test their spreadsheets incrementally by checking off that values in the spreadsheets are correct. The testing tool can use the user's validations to determine how much of the spreadsheet has been tested, and it can inform the user of the quality and coverage of the testing.

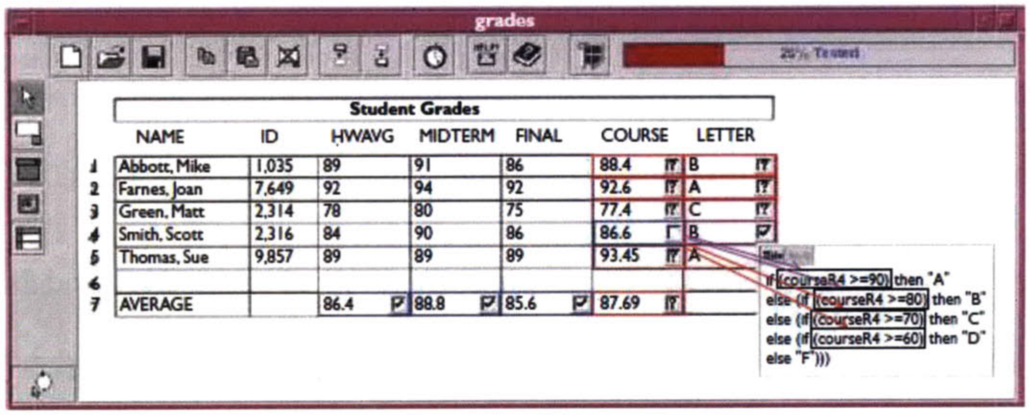
For example, in Figure 4, a teacher is using a spreadsheet to compute students' grades. The teacher can check off the checkboxes of cells with correct values. Checkboxes that are empty or have question marks indicate that the cell's value has not been checked, and question mark boxes will increase the testing coverage of the spreadsheet. Furthermore, the borders of the cells indicate how tested the cells are, based upon the dependencies between cells; bluer cells have been tested more thoroughly. This provides a visual feedback to the user as to which areas of the spreadsheet have been thoroughly tested and which could use more testing. The percentage bar at the top also conveys to the user what percentage of dependencies has been tested.



Student Grades

	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4 ?	B ?
2	Farnes, Joan	7,649	92	94	92	92.6 ?	A ?
3	Green, Matt	2,314	78	80	75	77.4 ?	C ?
4	Smith, Scott	2,316	84	90	86	86.6 ☐	B ✓
5	Thomas, Sue	9,857	89	89	89	93.45 ?	A ?
6							
7	AVERAGE		86.4 ✓	88.8 ✓	85.6 ✓	87.69 ?	

Figure 4. A teacher's spreadsheet with checkboxes to indicate that the cell values have been validated.



Student Grades

	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4	B
2	Farnes, Joan	7,649	92	94	92	92.6	A
3	Green, Matt	2,314	78	80	75	77.4	C
4	Smith, Scott	2,316	84	90	86	86.6	B
5	Thomas, Sue	9,857	89	89	89	93.45	A
6							
7	AVERAGE		86.4	88.8	85.6	87.69	

```
if (courseR4 >=90) then "A"
else (if (courseR4 >=80) then "B"
else (if (courseR4 >=70) then "C"
else (if (courseR4 >=60) then "D"
else "F")))
```

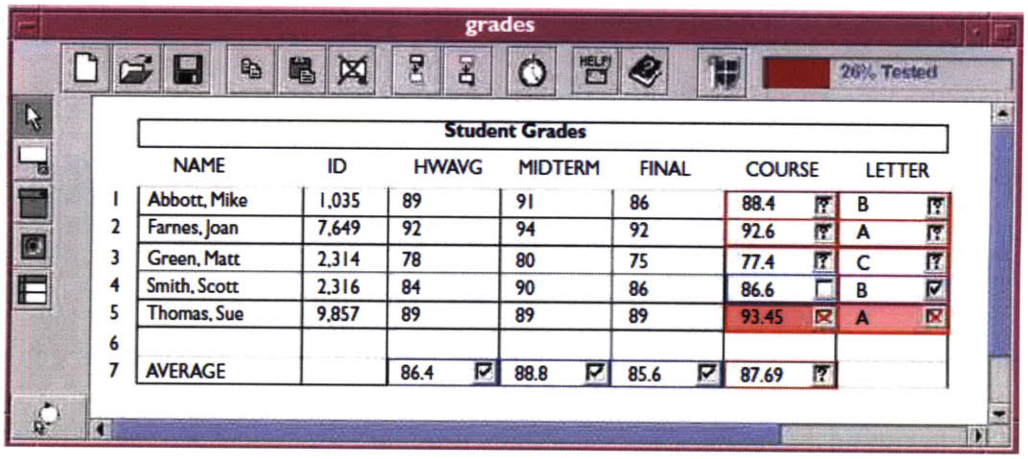
Figure 5. The spreadsheet shows dependencies between cells and how well tested they are.

To help users increase the testedness of their spreadsheets, the researchers included a Help-Me-Test button, which the user can click after selecting one or more cells [12]. The system tries to generate inputs that use dependencies that have not been tested yet. One way it can do this is by generating random inputs, propagating those values through the spreadsheet, and checking to see if it has used any dependency that has not been tested yet. The system then determines which output cells are relevant in this testing and can be validated, and it presents these cells to the user. The authors called this system Random. Seaweed's automated tester also uses this idea of generating random inputs, deriving an output, and having the user check that the output is correct.

They also developed a more complex testing system, which they call Chaining. Chaining does goal-oriented test case generation. Given a dependency (in particular, they use definition-use associations, which are the links between a cell with a defined value and a cell that eventually uses that defined value), the system finds the constraint path of associations that make up that dependency. It then tries to find inputs that use that constraint path and executes that dependency. During evaluation, it turned out that Chaining was much more effective and efficient than Random. However, if the Random method was given information about the valid ranges for its inputs, Random's effectiveness became much closer to Chaining's. Ko et al. note that visualization is also a useful technique for confirming the correctness of a program [19].

Debugging. Debugging is the process of determining the causes of known errors and fixing them. It is a very time-consuming process for both professional and end-user programmers [20][14][18]. In debugging, programmers start by asking why their programs behave in a certain, incorrect way, and they must figure out how to use low-level tools such as breakpoints and print statements to find the answer [15]. This difficult process is further complicated in the case of end users, who often do not possess a complete understanding of their program and thus have trouble hypothesizing the reasons for the incorrect behavior [16]. Additionally, end users who value achieving their goals over software reliability may try crude, makeshift fixes to their code, which may not solve the underlying problems and can end up generating even more bugs [17][3].

Burnett et al. also developed debugging tools for spreadsheets [9]. For example, in Figure 6, the Course and Letter grades for row 5 are incorrect. The user marks the cells as being incorrect. The system marks both of the cells with shaded pink backgrounds, but the Course cell is darker than the Letter cell. This is because the Course cell contributed to two errors (the course grade and the letter grade) whereas the Letter cell only contributed to one error (itself). The darker shading indicates a higher likelihood of Course containing a bug. As a visualization, the shading suggests to end users where to look first, and in user studies, the shading mechanism helped users be more effective at debugging.



Student Grades

	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4 ?	B ?
2	Farnes, Joan	7,649	92	94	92	92.6 ?	A ?
3	Green, Matt	2,314	78	80	75	77.4 ?	C ?
4	Smith, Scott	2,316	84	90	86	86.6 ☐	B ✓
5	Thomas, Sue	9,857	89	89	89	93.45 ☒	A ☒
6							
7	AVERAGE		86.4 ✓	88.8 ✓	85.6 ✓	87.69 ?	

Figure 6. Debugging mechanism for spreadsheets shows which cells contribute the most to errors.

Other end-user debugging tools are based on question asking. Users run their code and can press a button to ask why something happened or why didn't something happen [16]. The system then reviews the execution history of the program and gives users an answer based on that history. In user studies, this question asking technique has been very effective in reducing debugging time and improving debugging effectiveness [22].

Aside from these five major aspects of end-user software engineering, Ko et al. also discuss overarching issues in software engineering. When deciding whether or not to use a new software engineering tool, end users often deliberate on the perceived costs and benefits of the tool, as well as the expected pay-off and perceived risks. If they perceive new tools to be too risky or not worth their time, end users may become reluctant to learn about them and use them.

To encourage users to try new features, researchers have used an approach called Surprise-Explain-Reward [38][30][31]:

1. *Surprise* the user to generate curiosity about the feature.
2. *Explain* the feature and encourage the user to try the feature.
3. *Reward* the user for trying the feature and incentivize the user to use the feature again in the future

An example of the Surprise-Explain-Reward approach is how Burnett et al. encouraged users to use assertions in spreadsheets [9]. The researchers modified the Help-Me-Test feature to generate some assertions, in addition to generating the test case. The unexpected appearance of these assertions surprises the user. Tooltips explain the assertions feature to the user, and the visual feedback of assertions failing because of bugs or passing because of correctness rewards the user for trying out assertions. The execution of the Surprise-Explain-Reward in this scenario was careful not to disrupt the user's work routine. The feature passively seeks the user's attention. The user can choose to ignore the assertions; there are no dialog boxes to click, etc.

Other issues include self-efficacy, the individual's confidence in their ability to accomplish a specific task [2]. Low self-efficacy can cause a user to give up on a challenging software engineering task [7]. One study on a spreadsheet debugging task found that self-efficacy was correlated with how well females were able to use the debugging features, but that was not the case for males. The females had significantly lower self-efficacy and were less likely to try unfamiliar features than the males were, and so they were not able to debug as effectively [3].

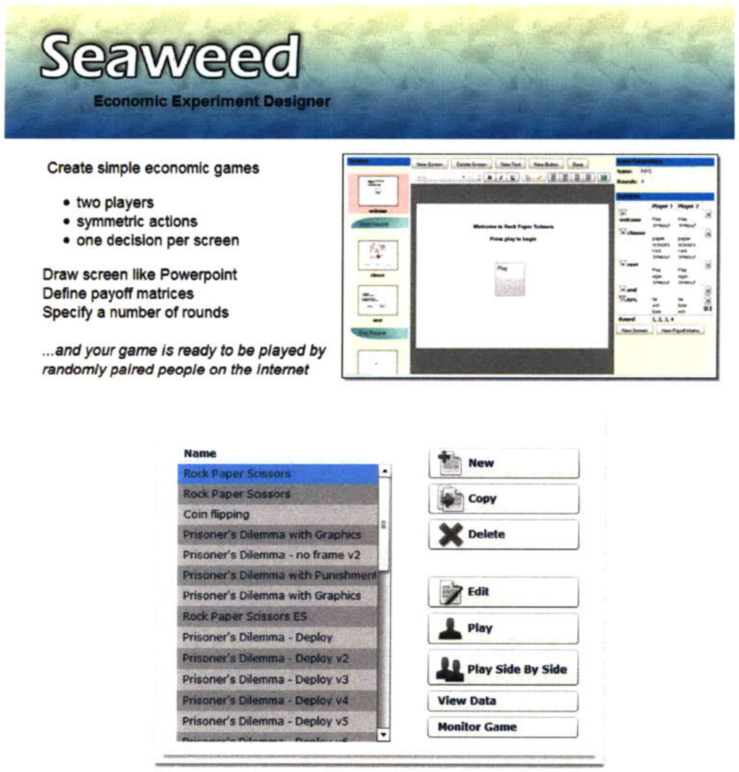
Another study looked at tinkering, or playful experimentation. Males tinkered more than females, but their tinkering was negatively related to success, whereas the females' tinkering was positively related to success [4]. The study attributed this to the fact that the females were more likely to pause between their actions, which gave them time to think about what they did and what effects their actions had.

These studies suggest that males and females encounter different kinds of difficulties when it comes to end-user software engineering, and there should be more research to understand and detect these barriers to success.

3. Seaweed

This thesis discusses the addition of the monitoring interface and the automated tester to Seaweed. To see how these features fit into Seaweed, we shall first describe how an economist uses Seaweed to create an economic game.

The Seaweed homepage (Figure 7) summarizes how an economist can use Seaweed. The page lists the Seaweed games that have been made. Buttons on the right give economists the power to make new games, copy existing games, and delete games. Users can also edit existing games, play them, use the Play Side By Side function (which allows a single user to play as both players of a game), view data, and monitor an ongoing game.



Create simple economic games

- two players
- symmetric actions
- one decision per screen

Draw screen like Powerpoint
Define payoff matrices
Specify a number of rounds

...and your game is ready to be played by randomly paired people on the internet

Name

Rock Paper Scissors
Rock Paper Scissors
Coin flipping
Prisoner's Dilemma with Graphics
Prisoner's Dilemma - no frame v2
Prisoner's Dilemma with Punishment
Prisoner's Dilemma with Graphics
Rock Paper Scissors ES
Prisoner's Dilemma - Deploy
Prisoner's Dilemma - Deploy v2
Prisoner's Dilemma - Deploy v3
Prisoner's Dilemma - Deploy v4
Prisoner's Dilemma - Deploy v5

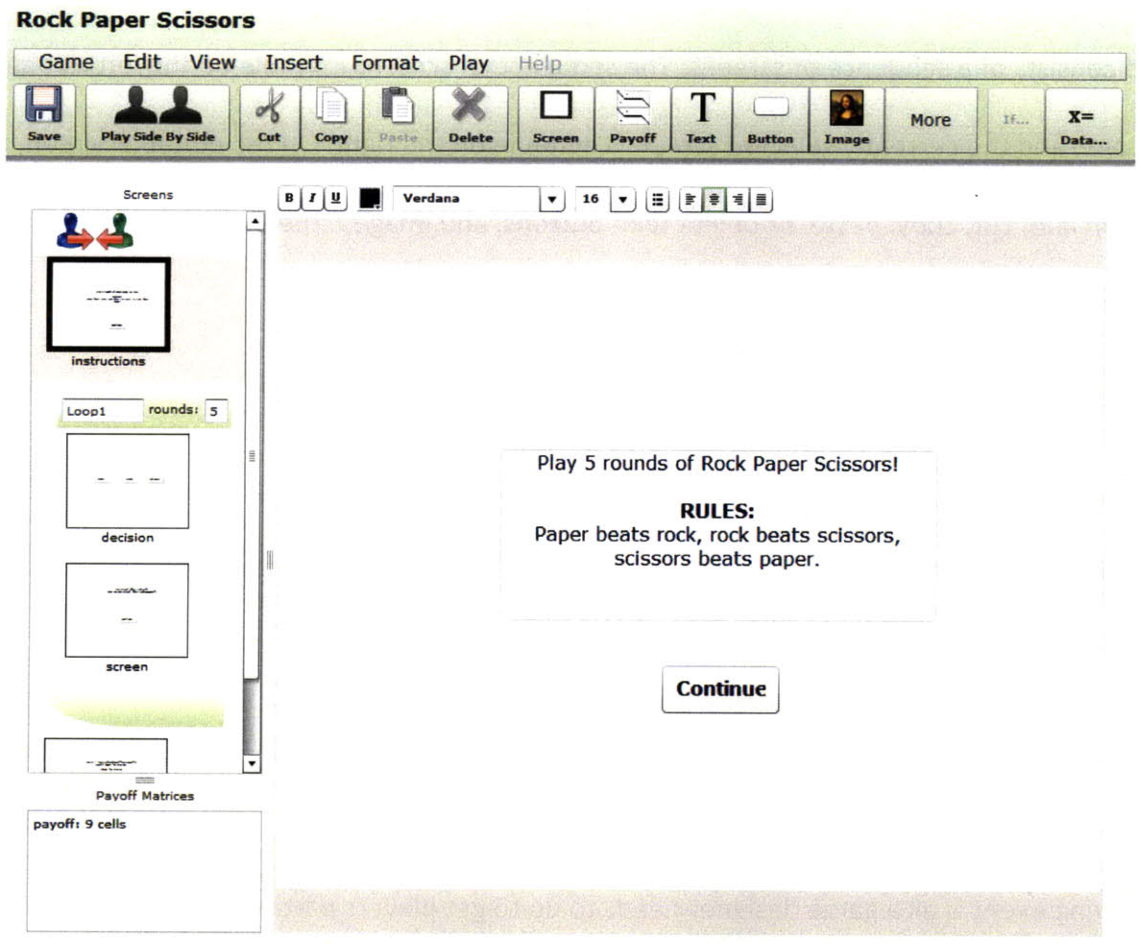
Figure 7. Seaweed homepage

Clicking on New, Copy, or Edit will bring up the game designer page (Figure 8). Every Seaweed game consists of a sequence of screens. The sequence of screens is visible on the left sidebar of the page, and the currently selected screen is editable on the right. On the editable screen, the user can add, cut, copy, paste, or delete text, buttons, and images. The user can also add more screens.

There are three other objects that can be added to a game: loops, partnering events, and payoff matrices. Loops are used to run multiple rounds of the same sequence of screens. For example, if a game designer wants the players to play five rounds of Rock Paper Scissors, the game could have a loop around the screens that need to be replayed.

Partnering events are put into the game at the point where it is necessary for players to be partnered up. Generally, players are partnered up at the beginning of the game. Inserting the partnering event is all a game designer needs to do to get players partnered up; Seaweed handles all the rest. This is one of the key advantages of Seaweed as a game designing platform for end users: the complicated details of how to handle mundane issues like partnering are hidden from the user, allowing the user to focus on the game design.

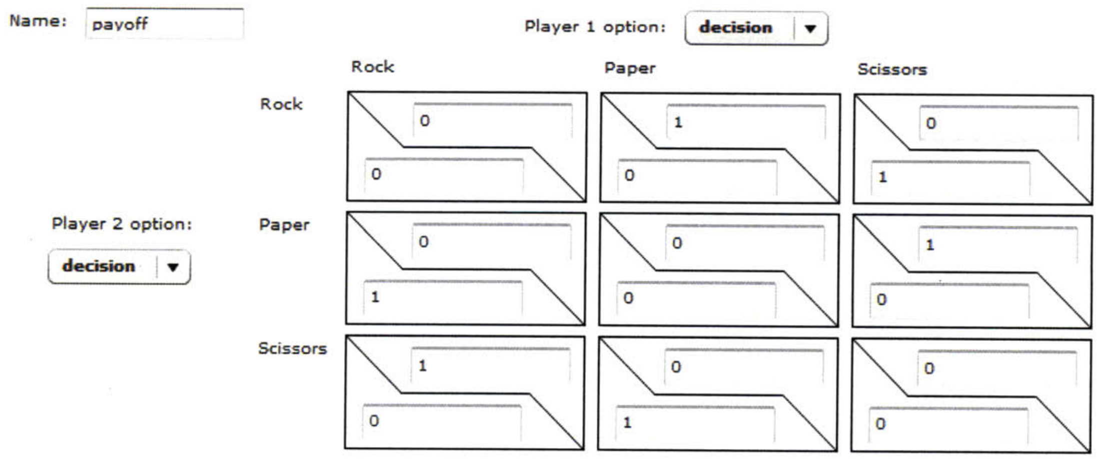
Payoff matrices are matrices that define what score or reward players receive, depending on their decisions and their partner's decisions. For example, in a Rock Paper Scissors game (Figure 9), a player can score a point if they choose Rock against the partner's Scissors, or Scissors against the partner's Paper, or Paper against the partner's Rock. Payoff matrices are the defining factor in many economic games, and they are used to give incentive to players to behave in a particular way.



Play 5 rounds of Rock Paper Scissors!

RULES:
Paper beats rock, rock beats scissors, scissors beats paper.

Figure 8. Game designer page for a Rock Paper Scissors game. The partnering event is visible above the screen named 'instructions' on the left sidebar. A loop of five rounds envelops the screens named 'decision' and 'screen.'



Name: payoff

Player 1 option: decision

Player 2 option: decision

	Rock	Paper	Scissors
Rock	0 / 0	1 / 0	0 / 1
Paper	0 / 1	0 / 0	1 / 0
Scissors	1 / 0	0 / 1	0 / 0

Figure 9. A payoff matrix for Rock Paper Scissors.

Once a user has designed and created a game, there are a few ways to test the game. One way is the Play Side By Side feature. With Play Side By Side, the designer can play as both players in a game, making decisions for both sides and checking to see if the game behaves as intended. This works very well, but it can take a long time to play through the game, and the designer does not have an easy way of viewing the data from multiple plays.

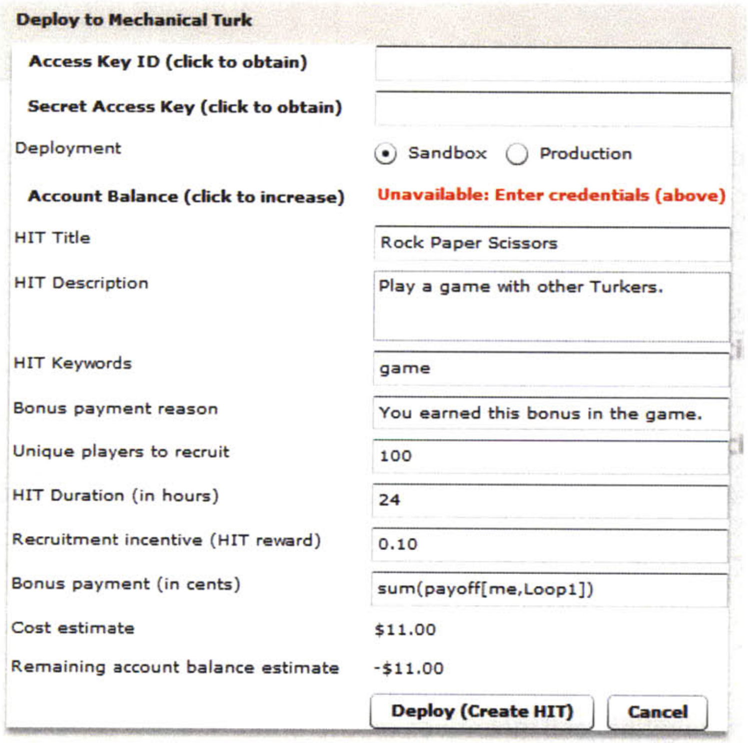
The other testing option is the automated tester, which runs multiple games and automatically makes random decisions for the players. It takes less time and manual effort than Play Side By Side, and it also shows the results of the multiple games in a human-readable table. However, because it is computer-run, it cannot test for things such as comprehensibility and clarity. The automated tester will be discussed in further detail later in this paper.

After the user is satisfied with the game, they will want to deploy the game so that people will play it. One way to do so would be to distribute the URL of the game. Players can go to the URL, get partnered up, and start playing the game. This can be used in a laboratory experiment setting. However, as discussed earlier, hiring subjects to come into the lab for an experiment can be expensive.

Seaweed offers an alternative form of deployment, using Amazon Mechanical Turk. Mechanical Turk is a crowdsourcing website that handles Human Intelligence Tasks, or HITs. HITs tend to be small jobs that are relatively easy for humans, but hard for computers. Requesters post HITs on Mechanical Turk, offering financial incentives on the order of a few cents per HIT. Workers accept and complete the HITs to earn the money.

A Seaweed user can deploy their game to Amazon Mechanical Turk very easily: they only have to fill out a short form with information such as their Amazon Access Key ID and Secret Access Key, a description of the HIT, the number of players to recruit, and the recruitment and bonus payments (Figure 10). This creates a *run*. The designer can make multiple runs of the same game, if they want to change the parameters or decide that they need more play data.

This deployment mechanism has been very successful. Because Mechanical Turk Workers will play a Seaweed game for cents, the cost of a Mechanical Turk deployment is much lower than that of a laboratory experiment.



Deploy to Mechanical Turk

Access Key ID (click to obtain)	
Secret Access Key (click to obtain)	
Deployment	[✓ Sandbox] Production
Account Balance (click to increase)	**Unavailable: Enter credentials (above)**
HIT Title	Rock Paper Scissors
HIT Description	Play a game with other Turkers.
HIT Keywords	game
Bonus payment reason	You earned this bonus in the game.
Unique players to recruit	100
HIT Duration (in hours)	24
Recruitment incentive (HIT reward)	0.10
Bonus payment (in cents)	sum(payoff[me,Loop1])
Cost estimate	$11.00
Remaining account balance estimate	-$11.00

Figure 10. Deployment to Mechanical Turk.

When a player starts playing a Seaweed game, Seaweed tries to create a partnership. Seaweed checks the partnership database to see if there are any players waiting to be partnered. If there is already a waiting player, Seaweed partners the two players together, and they start playing. Otherwise, Seaweed adds the new player to the partnership database, indicating that this player is waiting for a partner. There is the potential problem where Seaweed adds a player to the partnership database, the player does not get partnered for a long time and gives up playing the game, and when a new player finally arrives, they get paired with the player who has already left. Seaweed handles this problem by marking players in the partnership database as stale if it has been a while since they were put into the database.

There has been some research into more complex partnering algorithms than the naïve first-come, first-serve algorithm currently used by Seaweed. These algorithms require recruiting a large number of players at the same time to form a pool. From the pool, players are partnered with each other, and between rounds or games, they switch partners and are partnered with someone else from the pool. The algorithm determines how the players are partnered. Each algorithm is based on a graph, where the vertices represent the players and the edges represent allowed partnerships. For example, a ring graph (Figure 11) would mean that each player vertex has two neighbors or possible partners, and on each round, a player would be paired with one of those neighbors. On the next round, the player would be paired with the other neighbor, and on the next round, the player would be paired with the first neighbor again.

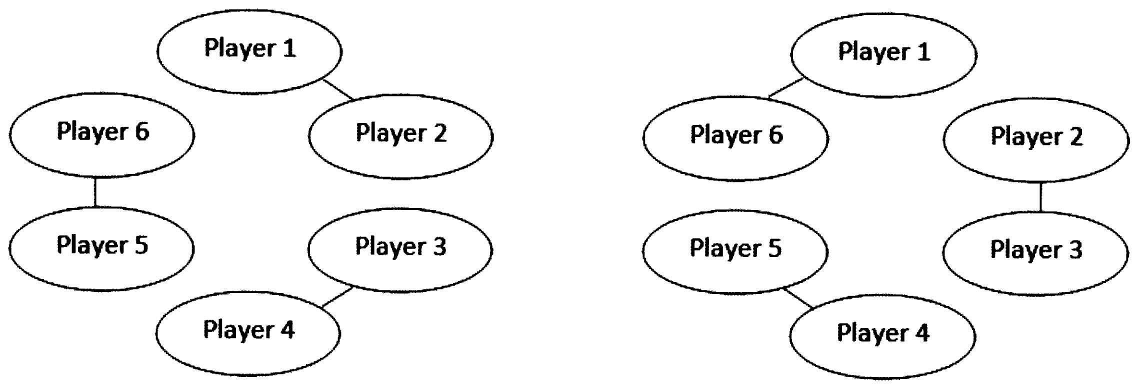


Figure 11. Partnering on a ring graph.

This sort of partnering algorithm creates a new situation where a player is not always playing with the same partner, but the player's action history is still important because they may be paired with an old partner again who remembers what that player has done in the past. This functionality is still under consideration and in development for Seaweed.

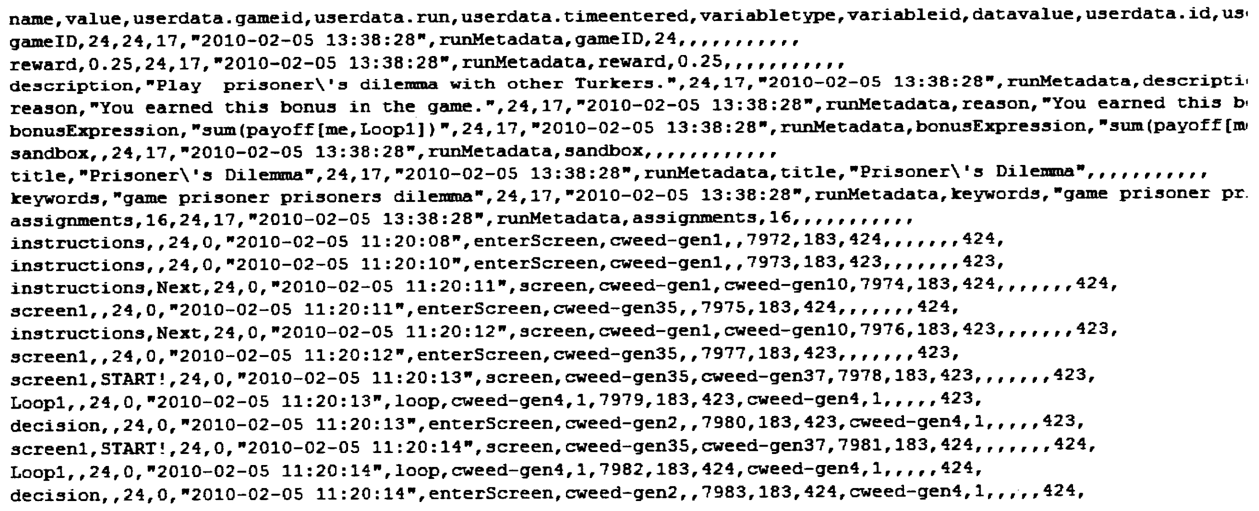
Once the two players are partnered up, each player starts playing the game. They can view and read whatever images or text was placed on the current screen. To move to the next screen, they must click a button. Sometimes, there will only be one button on the screen; this button merely acts as a way to get to the next screen. On screens with multiple buttons, the player is forced to make a decision about which button to click, a decision that may affect their score and ultimate payout.

Seaweed records all of the player's clicks in a database. This is not only for data collection, but it is also sometimes necessary for the game to function. Seaweed allows game designers to use players' decisions as values to be put into text or used in conditional statements. If a player reaches a screen whose text is dependent on the partner's decision and the partner has not made that decision yet, the player is forced to wait until the partner makes the decision. The values of evaluated payoff matrices are also available for the game designer to use. Payoff matrices are evaluated as soon as all the decisions for that matrix have been made.

Once a player has completed the game, they are paid the recruitment incentive. They are also paid the bonus payment, depending on their performance.

To view the play data, the game designer clicks on the View Data button on the Seaweed homepage. They are presented with a CSV file with all the play data for the selected game (Figure 12). Seaweed is liberal with the amount of play data that is recorded in the database, and the designer has access to read most of it. The designer can save the file and perform the desired analyses on the data.

Due to the sheer amount of data, however, the View Data feature is not appropriate for a real-time monitoring of an ongoing game. Instead, the designer clicks on the Monitor Game button and opens the monitoring interface, which provides a human-readable summary and view of any of the game's deployed runs.



```
name,value,userdata.gameid,userdata.run,userdata.timeentered,variabletype,variableid,datavalue,userdata.id,us
gameID,24,24,17,"2010-02-05 13:38:28",runMetadata,gameID,24,,,,,,,,,,,
reward,0.25,24,17,"2010-02-05 13:38:28",runMetadata,reward,0.25,,,,,,,,,,,
description,"Play  prisoner\'s dilemma with other Turkers.",24,17,"2010-02-05 13:38:28",runMetadata,descripti
reason,"You earned this bonus in the game.",24,17,"2010-02-05 13:38:28",runMetadata,reason,"You earned this b
bonusExpression,"sum(payoff[me,Loop1])",24,17,"2010-02-05 13:38:28",runMetadata,bonusExpression,"sum(payoff[m
sandbox,,24,17,"2010-02-05 13:38:28",runMetadata,sandbox,,,,,,,,,,,,
title,"Prisoner\'s Dilemma",24,17,"2010-02-05 13:38:28",runMetadata,title,"Prisoner\'s Dilemma",,,,,,,,,,,
keywords,"game prisoner prisoners dilemma",24,17,"2010-02-05 13:38:28",runMetadata,keywords,"game prisoner pr
assignments,16,24,17,"2010-02-05 13:38:28",runMetadata,assignments,16,,,,,,,,,,,
instructions,,24,0,"2010-02-05 11:20:08",enterScreen,cweed-gen1,,7972,183,424,,,,,,424,
instructions,,24,0,"2010-02-05 11:20:10",enterScreen,cweed-gen1,,7973,183,423,,,,,,423,
instructions,Next,24,0,"2010-02-05 11:20:11",screen,cweed-gen1,cweed-gen10,7974,183,424,,,,,,424,
screen1,,24,0,"2010-02-05 11:20:11",enterScreen,cweed-gen35,,7975,183,424,,,,,,424,
instructions,Next,24,0,"2010-02-05 11:20:12",screen,cweed-gen1,cweed-gen10,7976,183,423,,,,,,423,
screen1,,24,0,"2010-02-05 11:20:12",enterScreen,cweed-gen35,,7977,183,423,,,,,,423,
screen1,START!,24,0,"2010-02-05 11:20:13",screen,cweed-gen35,cweed-gen37,7978,183,423,,,,,,423,
Loop1,,24,0,"2010-02-05 11:20:13",loop,cweed-gen4,1,7979,183,423,cweed-gen4,1,,,,423,
decision,,24,0,"2010-02-05 11:20:13",enterScreen,cweed-gen2,,7980,183,423,cweed-gen4,1,,,,423,
screen1,START!,24,0,"2010-02-05 11:20:14",screen,cweed-gen35,cweed-gen37,7981,183,424,,,,,,424,
Loop1,,24,0,"2010-02-05 11:20:14",loop,cweed-gen4,1,7982,183,424,cweed-gen4,1,,,,424,
decision,,24,0,"2010-02-05 11:20:14",enterScreen,cweed-gen2,,7983,183,424,cweed-gen4,1,,,,424,
```

Figure 12. Part of the CSV file generated by the View Data feature

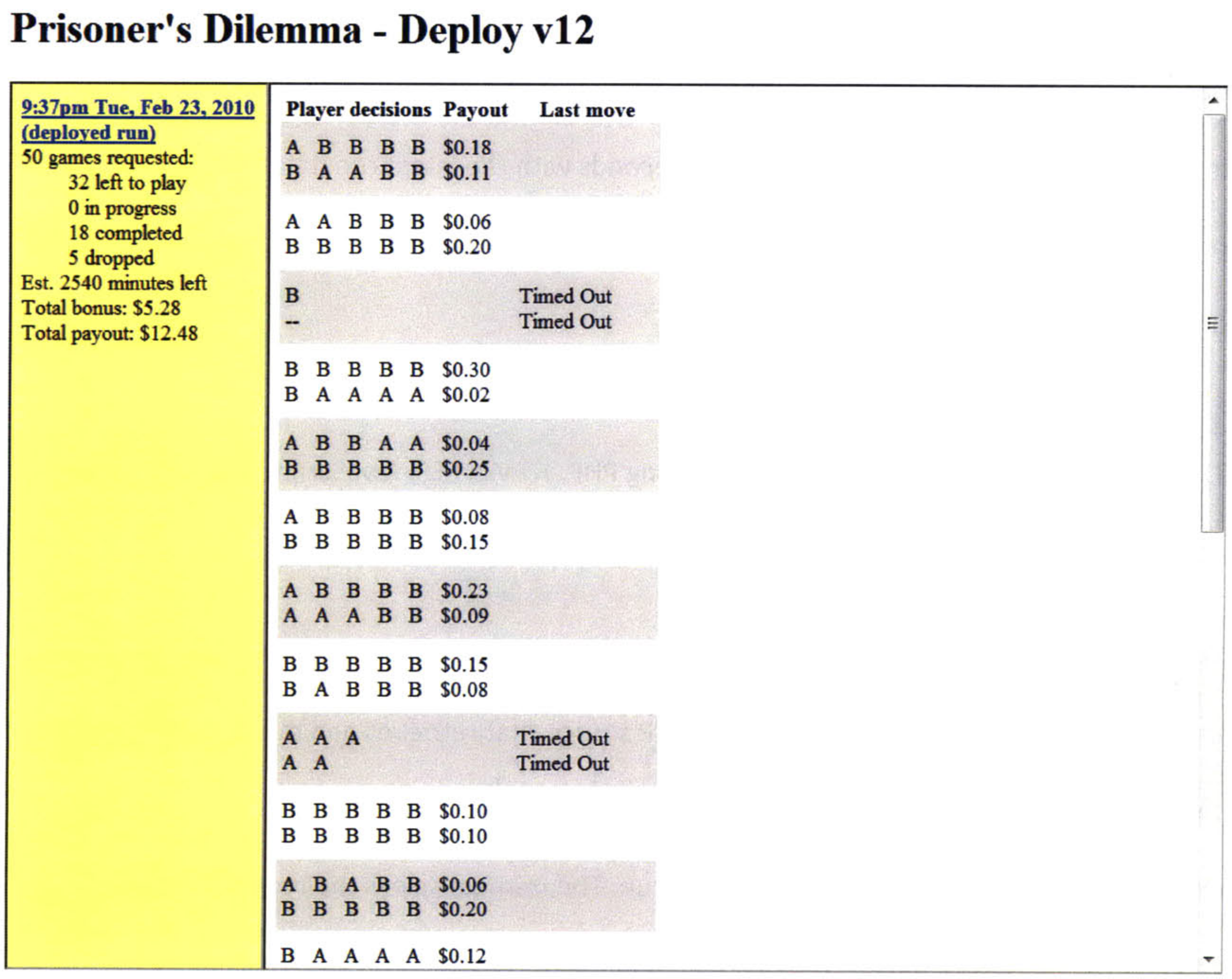
4. Monitoring Interface

The monitoring interface allows economists to observe the progress of a game deployment, also known as a run. Prior to the development of the monitoring interface, there was no reasonable way for the economists to watch a run in progress. The View Data feature allows economists reading access to all of the play data for a game, but the amount of data is overwhelming. The comma-separated format is well-suited for importing to spreadsheets and performing analysis, but it is difficult for humans to read directly. Lastly, View Data returns all of the play data for a game; it does not filter the data by which run the user may be interested in. View Data is more appropriate for use at the end of a run, to analyze all the data. To observe the play data during a run, the user should use the monitoring interface.

4.1 Design

The design of the monitoring interface was focused on one goal: the user must be able to easily and intuitively monitor the progress of all ongoing games in the run. There could be dozens of games in progress; the user has to be able to watch the important actions in each game without being overwhelmed by information.

To access the monitoring interface (Figure 13), the user selects a game on the Seaweed homepage and clicks on the Monitor Game button. The game's name is displayed prominently at the top. On the left is a sidebar that lists all the deployed runs for that game by date. By default, the most recent run is selected. The user can select a different run by clicking on the date link. The selected run's metadata is displayed in the sidebar. The user can easily tell how well the run is progressing by how many games are in progress, have been completed, were dropped or failed, or are left to play. The sidebar also gives a rough estimate of the amount of time needed to wait for the requested number of games to be completed, based on how quickly games are being completed. The user can also see how much bonus payment and total payment they have to pay so far.



Prisoner's Dilemma - Deploy v12

9:37pm Tue, Feb 23, 2010 (deployed run)
50 games requested:
32 left to play
0 in progress
18 completed
5 dropped
Est. 2540 minutes left
Total bonus: $5.28
Total payout: $12.48

Player decisions	Payout	Last move
A B B B B	$0.18	
B A A B B	$0.11	
A A B B B	$0.06	
B B B B B	$0.20	
B		Timed Out
--		Timed Out
B B B B B	$0.30	
B A A A A	$0.02	
A B B A A	$0.04	
B B B B B	$0.25	
A B B B B	$0.08	
B B B B B	$0.15	
A B B B B	$0.23	
A A A B B	$0.09	
B B B B B	$0.15	
B A B B B	$0.08	
A A A		Timed Out
A A		Timed Out
B B B B B	$0.10	
B B B B B	$0.10	
A B A B B	$0.06	
B B B B B	$0.20	
B A A A A	$0.12	

Figure 13. The Monitoring Interface.

The remainder of the page shows the summarized data for the run. There is a table with three columns: player decisions, payout, and last move. Aside from the header row, the rows are grouped into games, with each row representing a player. Games' rows are shaded in alternating gray and white, so that it is easy to distinguish games. Each row represents a player and displays their decisions, their final payout (if applicable), and how long ago their last move was. The interface only displays decisions that were made on screens with more than one button; decisions that were made on screens with only one button are trivial and unimportant. If it has been too long since a player made a move, the player has timed out, and the interface reflects this.

Because the monitoring interface's purpose is to allow the game designer to monitor a run, the interface automatically updates every five seconds with the latest run data. The newest player decisions are temporarily highlighted so the game designer can tell from a glance which of the ongoing games are still progressing.

4.2 Implementation

The monitoring interface is implemented using PHP, HTML, CSS, JavaScript, and AJAX. It also uses two JavaScript libraries, jQuery and Scriptaculous.

The monitoring page is generated by PHP and is comprised of the name of the game and two inline frames. The frame on the left has a PHP source that retrieves and displays all the runs for the game by their date of deployment. The user can select a run by clicking on its link; by default, the most recent run is the selected run. The frame displays the metadata for the selected run. The total bonus is the sum of all the bonus payouts given to players so far. The total payout is the recruitment payment times the number of players who are currently playing or have finished playing, plus the total bonus. Since players are not paid until they finish playing the game, and some players may drop out or get timed out, the total payout is sometimes an overestimate of the actual total payout.

The frame on the right displays the play data for the selected run. The PHP source fetches the play data for that run and sorts it by partnerships and players. It processes each player's play data and filters out the relevant data: the player's decisions on screens with more than one button, the bonus payout if applicable, and the time of the player's most recent action. If a player's decision was made in the last five seconds, it is marked so that later, we know we should highlight it as a new decision. The player's data is put into an array, and that array is inserted into an array of all the players' data. If a partnership is missing a player, an empty array for the missing player is inserted.

Once all of the run's play data has been processed, the program goes through the array of all the players' data and determines the maximum number of decisions any player has made. This is necessary to set up the HTML table which will eventually contain this data. The table is set up with the player decisions, payout, and last move header cells. The player decisions header cell spans the maximum number of decisions for any player. Each player corresponds to a row, and each partnership corresponds to a shaded or unshaded contiguous block of rows. In each row, the player's decisions are put into individual cells; if the player has not made the maximum number of decisions yet, placeholder cells are inserted. The bonus payout (if applicable) is also added to the row. The time of the last move is also added to the row, displayed as how many seconds it has been since then (e.g. "15s ago"). If it has been too long since the last move (i.e. 120 seconds), we assume the player has timed out, and the last move column displays "Timed Out".

As mentioned earlier, a partnership that is missing a player will have an empty array. When this empty array is encountered, a row is created for the missing player with only the last move cell displaying the message “awaiting partner”.

We use AJAX and jQuery to automatically update the page with new data every five seconds. For each frame, every five seconds, a call is made to the appropriate PHP source script, and the output is loaded into a div in the frame.

To highlight the latest player decisions, we use a JavaScript library called Scriptaculous, which adds dynamic visual effects to web applications. When the div is updated by the AJAX call, we use Scriptaculous to highlight the table cells that were marked for highlighting. The ultimate effect is that when the newest player decisions appear, they are highlighted in yellow, and the highlight slowly fades away.

5. Automated Testing

The automated testing function allows economists to automatically generate play data for their game. The economists can then review the data and determine if the results are as expected and the bonus payout is being computed properly. The automated tester can also detect when there are deadlocks in the game.

5.1 Computer Player

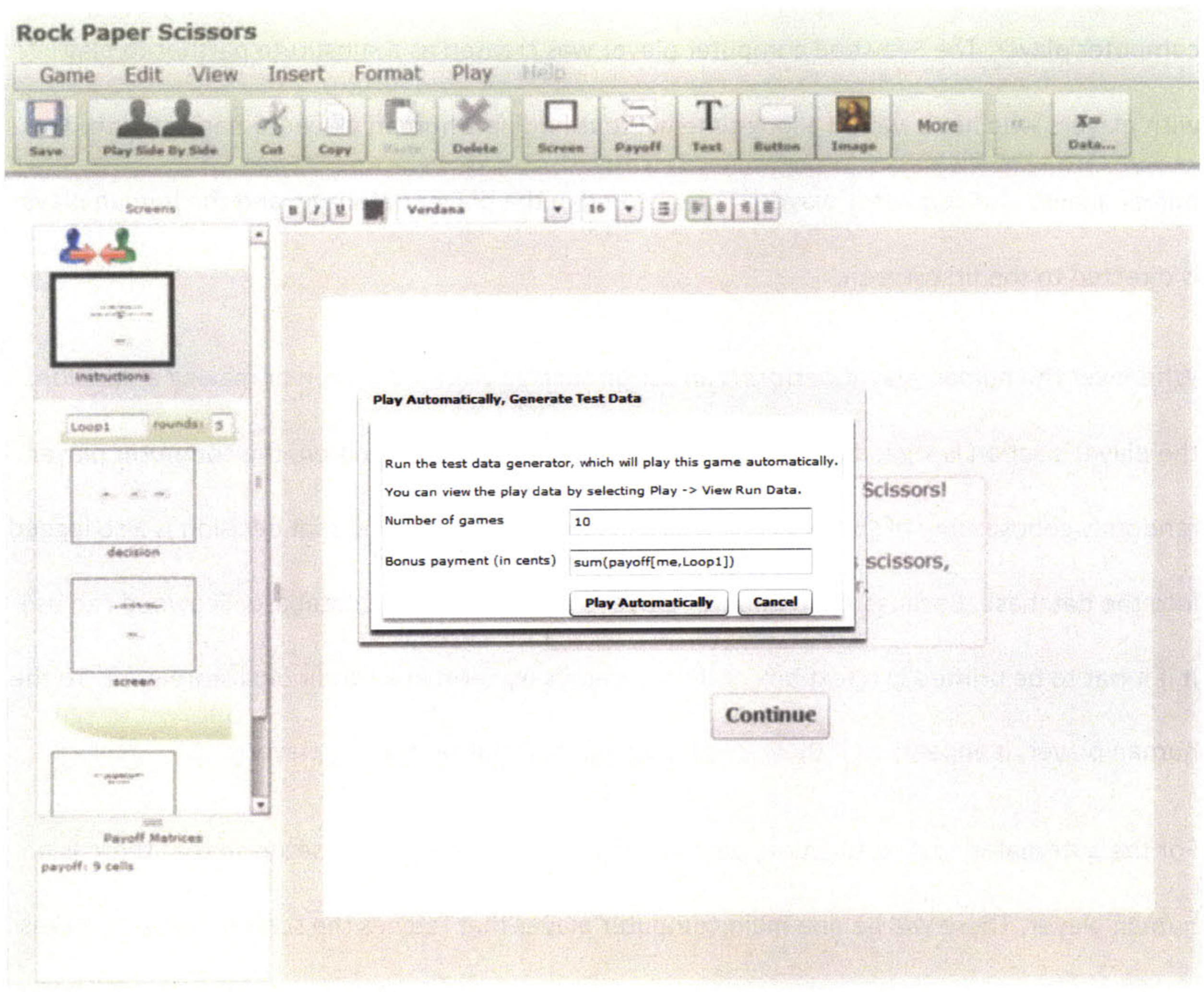
The development of the automated testing function grew out of previous work on the Seaweed computer player. The Seaweed computer player was created as a substitute partner to play with players who have waited a long time without finding a human player to partner with. The human player and computer player are registered in the player database, and the human player is directed to the first screen.

Whenever the human player performs an action such as visiting a screen or making a decision, the player's action is logged into the database. If that action is a decision, the computer player randomly chooses one of the available decisions for that screen, and that decision is also logged into the database. Because the computer player's decision is in the database, Seaweed can use it if it has to be printed in a text box or if it's a necessary term in a conditional statement. To the human player, it appears as if they are playing against another human player.

For the automated testing function, we modified the computer player setup so that there is no human player. There will be one main computer player that fetches the screen data and makes the first random decision, and the remaining computer player follows and makes a random decision whenever the main computer player does.

5.2 Design and Implementation

To run the automated testing feature, the user accesses the game design page, selects the "Play" menu, and clicks on "Play Automatically..." A small form appears, explaining the automated testing feature (Figure 14). It asks the user for the number of games to be played automatically and for the bonus payment expression.



Play Automatically, Generate Test Data

Run the test data generator, which will play this game automatically.

You can view the play data by selecting Play -> View Run Data.

Number of games	10
Bonus payment (in cents)	sum(payoff[me,Loop1])

Figure 14. Automated Testing Form.

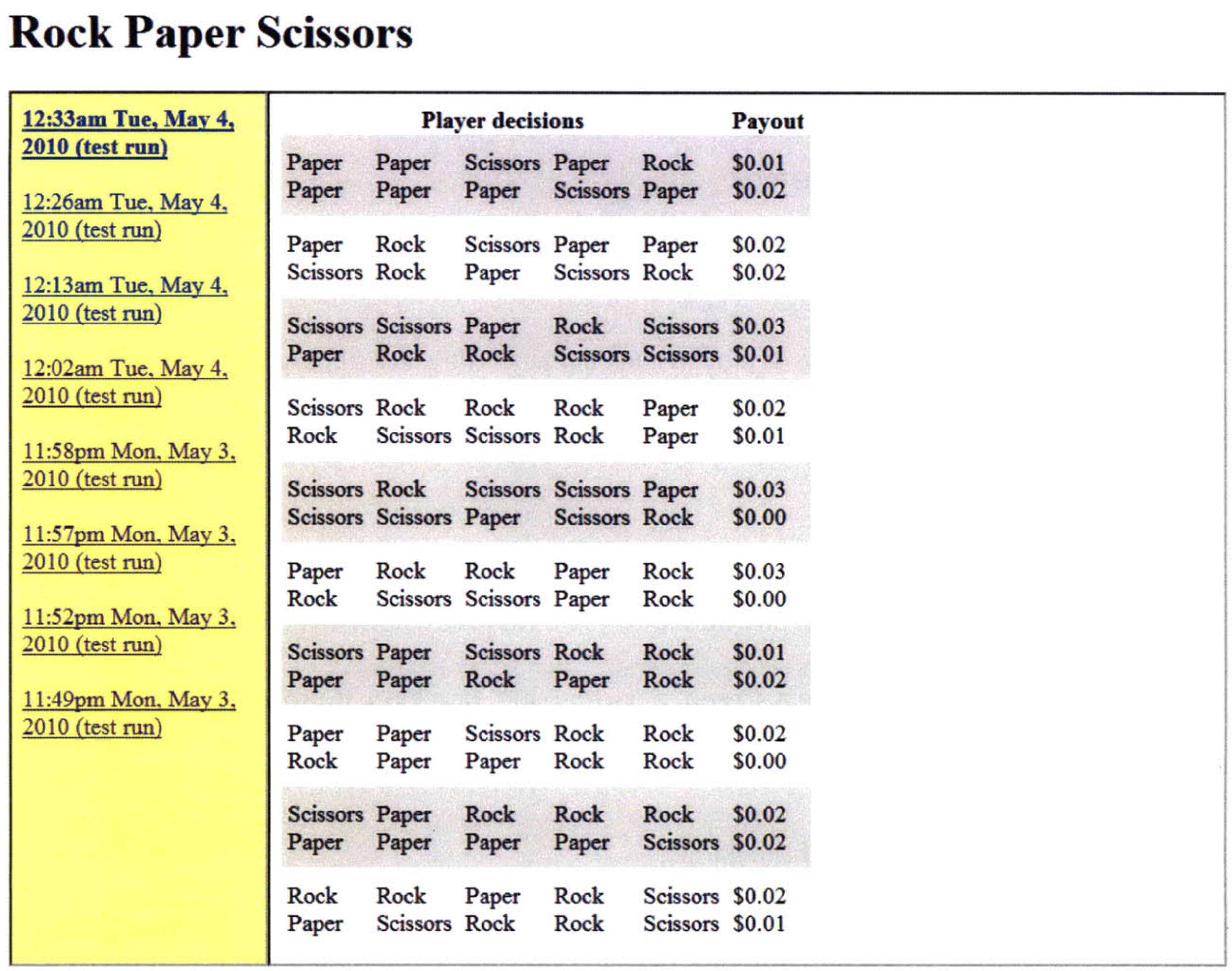
When the user clicks on the "Play Automatically" button, a new window opens and displays the monitoring interface described earlier in this thesis. The monitoring interface has a second role: to display the results of the test games.

In the background, a PHP script is generating the test data. It creates a run for the test data. Test runs are distinguishable from deployed runs because they lack deployment settings such as duration of the run and recruitment payment. The script then runs the test games, one at a time.

First, a computer player is created and added to the database. The script then selects the first node of the game, which could be a screen, a partnership, or a loop. It processes the node, finds the next node, and continues processing and finding nodes until it has run out of nodes because it has reached the end of the game.

Usually, the first node is a partnership formation. We create as many computer player partners as necessary and add them to the player database. (Seaweed is currently limited to two-player games. However, many sections of Seaweed, such as this one, are designed to ease the transition to n-player games, should that change be implemented in the future.)

For screen nodes, the tester checks the conditional and textual dependencies of the screen. It's possible that a dependency is not satisfied; for example, a block of text on the third screen tries to output the decision made on the fourth screen, but it cannot because that decision has not been made yet. In that case, the tester deduces that there is a deadlock, logs a deadlock message, and terminates the testing for this test game.



Rock Paper Scissors

12:33am Tue, May 4, 2010 (test run)

12:26am Tue, May 4, 2010 (test run)

12:13am Tue, May 4, 2010 (test run)

12:02am Tue, May 4, 2010 (test run)

11:58pm Mon, May 3, 2010 (test run)

11:57pm Mon, May 3, 2010 (test run)

11:52pm Mon, May 3, 2010 (test run)

11:49pm Mon, May 3, 2010 (test run)

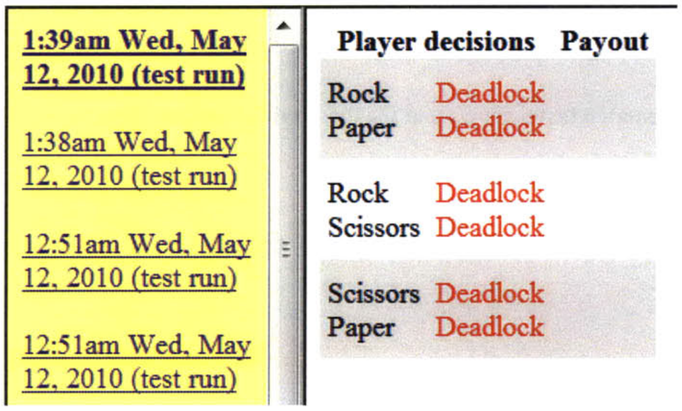
Player decisions					Payout
Paper	Paper	Scissors	Paper	Rock	$0.01
Paper	Paper	Paper	Scissors	Paper	$0.02
Paper	Rock	Scissors	Paper	Paper	$0.02
Scissors	Rock	Paper	Scissors	Rock	$0.02
Scissors	Scissors	Paper	Rock	Scissors	$0.03
Paper	Rock	Rock	Scissors	Scissors	$0.01
Scissors	Rock	Rock	Rock	Paper	$0.02
Rock	Scissors	Scissors	Rock	Paper	$0.01
Scissors	Rock	Scissors	Scissors	Paper	$0.03
Scissors	Scissors	Paper	Scissors	Rock	$0.00
Paper	Rock	Rock	Paper	Rock	$0.03
Rock	Scissors	Scissors	Paper	Rock	$0.00
Scissors	Paper	Scissors	Rock	Rock	$0.01
Paper	Paper	Rock	Paper	Rock	$0.02
Paper	Paper	Scissors	Rock	Rock	$0.02
Rock	Paper	Paper	Rock	Rock	$0.00
Scissors	Paper	Rock	Rock	Rock	$0.02
Paper	Paper	Paper	Paper	Scissors	$0.02
Rock	Rock	Paper	Rock	Scissors	$0.02
Paper	Scissors	Rock	Rock	Scissors	$0.01

Figure 15. Monitoring Interface for viewing test data.

If there is no deadlock, all the computer players make a random decision and select a button. This decision is logged. Loop nodes are processed exactly as they are for human players.

Once all of the nodes are processed, the bonus payout is computed from the bonus payout expression entered by the user. The bonus payout is computed for both the main computer player and its opponent, and the payouts are inserted into the database.

While the script is running, the user is viewing the monitoring interface (Figure 15). Because the test run is the most recent run, by default, it is the one displayed on the interface. The user watches as the results of the test data generation are revealed on the page. At this point, the user can check the results and confirm that the player decisions are appropriate and that the bonus payouts match the user's expectations. If the bonus payouts do not match the user's expectations, this should encourage the user to review the game design to make sure that everything is written and implemented as intended. If there is a deadlock, the interface produces a deadlock error message. The error message is placed in the player decision sequence at the location of the deadlock. For example, in Figure 16, the tester was able to make the first decision, but deadlock prevented any subsequent decisions.



1:39am Wed, May 12, 2010 (test run)

1:38am Wed, May 12, 2010 (test run)

12:51am Wed, May 12, 2010 (test run)

12:51am Wed, May 12, 2010 (test run)

Player decisions	Payout
Rock	Deadlock
Paper	Deadlock
Rock	Deadlock
Scissors	Deadlock
Scissors	Deadlock
Paper	Deadlock

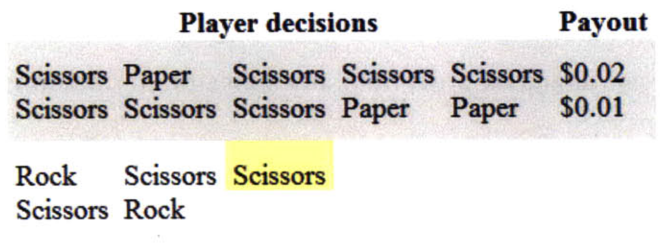
Figure 16. Deadlock message.

There are a few small differences between how test runs and deployed runs are displayed in the interface. Test runs do not have any run metadata, so none of that information is displayed in the left frame. Additionally, because test runs are generated by the server running as quickly as possible, there is no need for a "last move" column.

5.3 Testing the Monitoring Interface

An interesting benefit of the automated tester is that with small modifications, it can be used to test the monitoring interface. By adding small time delays to the computer players' actions, we can simulate human players playing the game. The alternative would be to deploy runs to Amazon Mechanical Turk, which may cost money and takes time to recruit players.

By using the automated tester, we were able to confirm that the monitoring interface does update every five seconds as intended and that new player decisions are highlighted.



		Player decisions			Payout
Scissors	Paper	Scissors	Scissors	Scissors	$0.02
Scissors	Scissors	Scissors	Paper	Paper	$0.01
Rock	Scissors	Scissors			
Scissors	Rock				

Figure 17. A modified automated tester shows that the monitoring interface is highlighting new decisions properly.

6. Conclusion

Seaweed is a web-based economic game system for end users. Economists can use Seaweed to design and deploy simple two-player economic games. To improve Seaweed's usability, we added the monitoring interface and automated testing features.

The monitoring interface allows users to monitor their deployed runs. The users can observe players' decisions in real time. The interface also shows when each player last performed an action and what the player's bonus payout is. The interface uses AJAX to update the page automatically. The sidebar allows the user to track how many games have been successfully completed, are currently being played, have failed and been left uncompleted, or still need to be played. It also gives a rough estimate of the time needed to complete all the games and displays the total payout for the run so far.

The automated testing feature allows users to automatically play multiple games. The automated tester simulates the play of both players of a game. When a player must make a decision, the tester chooses a decision randomly. It records all the decisions of each game and computes the bonus payouts associated with those decisions. The user can then view the results and check that the results are as expected. The tester also detects deadlocks and alerts the user of their existence.

With the addition of these features, we have improved the usability of Seaweed. The monitoring interface gives economists immediate feedback on their deployed runs and how well they are progressing. If they use it in a laboratory setting, it may help them identify problems, such as many players being unable to complete their games. The automated tester gives economists confidence that their games are working properly. By seeing simulated runs, the economists can check that there are no deadlocks, the bonus payouts are being computed correctly, and the simulated players are making appropriate decisions. Thus, they may feel more comfortable when the time comes for deployment.

7. Future Work

There is much potential for improvement and expansion of the Seaweed features described in this thesis.

For both the monitoring interface and the automated tester, we have not performed any user studies with the economists that we were working with. This means that any conclusions we make about the usefulness of these features are drawn from our own use of the features and our predictions of how end users will use the system. User studies would provide valuable feedback as to how these features can be improved.

The automated tester currently runs on the server side as a PHP script. Ideally, the tester would simulate client-side actions, because those are what human players do. By simulating client-side play, the tester would notice errors such as race conditions. While there may not be anything an end user can be reasonably expected to do about race conditions, client-side simulations may be helpful for Seaweed developers who want to test for race conditions.

The deadlock detection mechanism in the automated tester is effective, but it could be improved. It indicates where the deadlock is relative to the players' non-trivial decisions. However, we can imagine a game with a screen that uses many data values, and although the user has determined that this is the screen responsible for the deadlock, it is not clear which data value is the culprit. Extending the deadlock detection mechanism to pinpoint exactly which data dependence was responsible can help users fix deadlocks more easily.

Another possible improvement for the automated tester would be to add checkboxes for the user to check off the games that produced the expected bonus payout. This would be similar to the checkboxes used for testing spreadsheets, and it may have the same benefits. Additionally, we could determine which payoff matrices or which values contributed to the bonus payout, so that in the case where there are some games that are behaving correctly and some that are not, we may be able to direct the user to the most likely causes of the error. It may also be beneficial to display the results of evaluated payoff matrices, but it may also lead to a more cluttered results page, so it would require careful design and user studies to determine its effectiveness.

8. References

[1] Abraham, R. and M. Erwig. "Inferring templates from spreadsheet." International Conference on Software Engineering. Shanghai, China, 2006. 182-191.

[2] Bandura, A. "Self-efficacy: Toward a unifying theory of behavioral change." Psychological Review (1977): 191-215.

[3] Beckwith, L., et al. "Effectiveness of end-user debugging features: Are there gender issues?" ACM Conference on Human Factors in Computing Systems. Portland, OR, 2005. 869-878.

[4] Beckwith, L., et al. "Tinkering and gender in end-user programmers' debugging." ACM Conference on Human Factors in Computing Systems. Montreal, Canada, 2006. 231-240.

[5] Belletini, C., E. Damiani and M. Fugini. "User opinions and rewards in reuse-based development system." Symposium on Software Reusability. Los Angeles, CA, 1999. 151-158.

[6] Bellon, S., et al. "Comparison and evaluation of clone detection tools." IEEE Transactions on Software Engineering (2007): 577-591.

[7] Blackwell, A. F., J. A. Rode and E. F. Toye. "How do we program the home? Gender, attention investment, and the psychology of programming at home." International Journal of Human Computer Studies (2009): 324-341.

[8] Bogart, C., et al. "End-user programming in the wild: A field study of CoScripter scripts." IEEE Symposium on Visual Languages and Human-Centric Computing. 2008.

[9] Burnett, Margaret, Curtis Cook and Gregg Rothermel. "End-User Software Engineering." Communications of the ACM September 2004: 53-58.

[10] Chilton, Lydia. Seaweed: A Web Application for Designing Economic Games. Cambridge, MA: Massachusetts Institute of Technology, 2009.

[11] Erwig, M., et al. "Automatic generation and maintenance of correct spreadsheets." International Conference on Software Engineering. St. Louis, MO, 2005. 136-145.

[12] Fisher, Marc, et al. "Automated Test Case Generation for Spreadsheets." Proceedings of the 24th International Conference on Software Engineering. ACM Press, 2002. 141-151.

[13] Hendry, D. G. and T. R. G. Green. "Creating, comprehending, and explaining spreadsheets: A cognitive interpretation of what discretionary users think of the spreadsheet model." International Journal of Human-Computer Studies (1994): 1033-1065.

[14] Ko, A. J. and B. A. Myers. "A Framework and methodology for studying the causes of software errors in programming systems." Journal of Visual Languages and Computing (2005): 41-84.

[15] Ko, A. J. and B. A. Myers. "Debugging reinvented: Asking and answering why and why not questions about program behavior." International Conference on Software Engineering (ICSE). Leipzig, Germany, 2008. 301-310.

[16] Ko, A. J. and B. A. Myers. "Designing the Whyline: A debugging interface for asking questions about program failures." ACM Conference on Human Factors in Computing Systems. Vienna, Austria, 2004. 151-158.

[17] Ko, A. J. and B. A. Myers. "Development and evaluation of a model of programming errors." IEEE Symposium Human-Centric Computing Languages and Environments. Auckland, New Zealand, 2003. 7-14.

[18] Ko, A. J., R. Deline and G. Venolia. "Information needs in collocated software development teams." International Conference on Software Engineering. Minneapolis, MN, 2007. 344-353.

[19] Ko, Andrew J, et al. "The State of the Art in End-User Software Engineering." ACM Computing Surveys (2009).

[20] Latoza, T., G. Venolia and R. Deline. "Maintaining mental models: A study of developer work habits." International Conference on Software Engineering. Shanghai, China, 2006. 492-501.

[21] Leiberman, H. Your Wish is My Command: Giving Users the Power to Instruct their Software. San Francisco, CA: Morgan Kaufmann, 2000.

[22] Lim, B. and D. Avrahami. "Why and why not explanations improve the intelligibility of context-aware intelligent systems." ACM Conference on Human Factors in Computing Systems. Boston, MA, 2009. 2119-2128.

[23] Mandelin, D., et al. "Jungloid mining: Helping to navigate the API jungle." ACM SIGPLAN Conference on Programming Language Design and Implementation. Chicago, IL, 2005. 48-61.

[24] Marick, Brian. "When Should a Test Be Automated?" STAREAST. 1999.

[25] McDaniel, R. and B. Myers. "Getting more out of programming-by-demonstration." ACM Conference on Human Factors in Computing Systems. Pittsburg, PA, 1999. 442-449.

[26] Panko, R. "Spreadsheet errors: what we know. what we think we can do." Spreadsheet Risk Symposium. 2000.

[27] Panko, R. "What we know about spreadsheet errors." Journal of End User Computing (1998): 15-21.

[28] Phalgune, A., et al. "Garbage in, garbage out? An empirical look at oracle mistakes by end-user programmers." IEEE Symposium on Visual Languages and Human-Centric Computing. Dallas, TX, 2005. 45-52.

[29] Ravichandran, T. and M. Rothenberger. "Software reuse strategies and component markets." Communications of the ACM (2003): 109-114.

[30] Robertson, T. J., et al. "Impact of interruption style on end-user debugging." ACM Conference on Human Factors in Computing Systems. Vienna, Austria, 2004. 287-294.

[31] Ruthruff, J. R., et al. "Rewarding "good" behavior: End-user debugging and rewards." IEEE Symposium on Visual Languages and Human-Centered Computing. Rome, Italy, 2004. 115-122.

[32] Ruthruff, J., et al. "Interactive, visual fault localization support for end-user programmers." Journal of Visual Languages and Computing (2005): 3-40.

[33] Ruthruff, J., M. Burnett and G. Rothermel. "An empirical study of fault localization for end-user programmers." International Conference on Software Engineering. St. Louis, MO, 2005. 352-361.

[34] Scaffidi, C., B. A. Myers and M. Shaw. "Reusable abstractions for validating data." International Conference on Software Engineering. Leipzig, Germany, 2008. 1-10.

[35] Scaffidi, C., M. Shaw and B. A. Myers. "Estimating the number of end users and end user programmers." IEEE Symposium on Visual Languages and Human-Centric. Brighton, UK, 2005. 207-214.

[36] Stylos, J. and B. A. Myers. "Mica: A web-search tool for finding API components and examples." IEEE Symposium on Visual Languages and Human-Centric Computing. Brighton, UK, 2006. 195-202.

[37] Wilcox, E., et al. "Does continuous visual feedback aid debugging in direct-manipulation programming systems?" ACM Conference on Human Factors in Computing Systems. Atlanta, GA, 1997. 258-265.

[38] Wilson, A., et al. "Harnessing curiosity to increase correctness in end-user programming." ACM Conference on Human Factors in Computing Systems. Fort Lauderdale, FL, 2003. 305-312.

[39] Ye, Y. and G. Fischer. "Reuse-conducive development environments." International Journal of Automated Software Engineering (2005): 199-235.