UNIVERSITY OF CALIFORNIA
RIVERSIDE

# Analysis, Design, Development, and Deployment of a Generalized Framework for Computer-Aided Assessment

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science
in
Computer Science
by
Titus Delafayette Winters
June 2004

Thesis Committee:

Dr. Tom Payne, Chairperson
Dr. Mart Molle
Dr. Christian Shelton

The Thesis of Titus Delafayette Winters is approved:

_____

_____

_____

Committee Chairperson

University of California, Riverside

# Acknowledgments

I would like to acknowledge Dave Sheldon and Keri Nishimoto, the most important users of Agar: your input was invaluable. I would like to thank the undergrads who have contributed to the project: Eric Harris, Steve Bui, Ed Levie, and Vinh Lam. I would like to thank Dr. Geoff Kuenning, who gave me my first grading job. I would like to thank Dan Berger for his input on a draft of this document and continual wisdom regarding development. Most importantly, I would like to thank my advisor, Dr. Tom Payne, for his support on what once seemed like a small project.

ABSTRACT OF THE THESIS

Analysis, Design, Development, and Deployment of a Generalized Framework for Computer-Aided Assessment

by

Titus Delafayette Winters

Master of Science, Graduate Program in Computer Science
University of California, Riverside, June 2004
Dr. Tom Payne, Chairperson

Over the past year the author has experimented with various approaches to Computer-Aided Assessment (CAA) ranging from custom shell-scripts for grading assignments to a complex GUI-based framework capable of handling Optical Mark Recognition and subjective grading of essay questions. The pros and cons of each approach are presented, focusing on barriers to adoption, level of tolerance to unexpected submission behavior, applicability in non-programming domains, and required user competency. An analysis of design requirements for a sufficiently general framework for CAA, based on the author's development experience, is presented, followed by a discussion of a system built to meet those requirements – called Agar – and improvements planned for Agar2.

# Contents

# List of Figures

# Chapter 1

# Introduction

Computer-aided assessment (often referred to as automated grading or simply CAA) is an idea that surfaces in many (if not most) computer science departments in one form or another. In some academic institutions, CAA manifests itself as a custom script to help ensure some consistency across graders. For example, in the author's undergraduate career, grading for the Harvey Mudd College C++ and Data Structures course was done with the aid of a script written for each assignment by the professor in charge that made a rough estimate of the functional correctness of a student submission followed by a series of prompts regarding the level of style and documentation in the submission.

On the opposite end of the spectrum, CAA is commonly used with objective assessment items like multiple choice questions (MCQs), matching, and other computer-graded/discrete-response forms of assessment [11]. Indeed, CAA forms the basis for all (or nearly all) of the standardized testing that is done in the US, from elementary school standardized tests where optical mark recognition is used to evaluate multiple-choice question (MCQ) responses, to Item Response Theory [7]-based computer-adaptive tests, like the GRE, where a computer program adapts to the student's responses in an attempt to provide a more precise score for test takers.

Nevertheless, relatively little [19] [16] work has been done on the development of a generalized and *usable* framework for CAA until now. CAA offers the ability to reduce repetition, vanquish clerical error, and increase human time efficiency. This work is greatly motivated by the author's experiences with the development of Agar, a prime[1] example of a generalized framework for CAA guided by principles gained from the domain of Human-Computer Interfaces (HCI). This work is not an advancement in the field of HCI, and as shown in Chapter 3, CAA has a long history of attempts to come up with a magic bullet for grading. What is important in this work is that we have allowed the development of Agar to be guided by HCI and such classic notions on Software Engineering as Fred Brooks' "The Pilot System."[9]. The lessons learned, insight gained, and user feedback gathered during the development and deployment of Agar has given us a very clear understanding of what is necessary to create a highly usable CAA tool.

The remainder of this document will discuss the design, development, and deployment of Agar, the system for Computer Assisted Assessment developed in the CS&E Department at UCR over the past year. As discussed in Chapter 6, this process was an "organic" variant on the spiral model: no clear notion of interface design or final feature set existed when development began. Rather, development was guided by the needs of users in a series of very short cycles of feature addition, interface alteration, and bug fixing.

[1] And as of this writing, the only known.

# Chapter 2

# Learning and Assessment

## 2.1 Constructivism

Much of the following discussion on what constitutes helpful grading is influenced by the vogue pedagogical philosophy of constructivism[2], a theory of understanding introduced by Jerome Bruner[10]. The constructivist view of education is that learning is primarily a search for meaning that is accomplished by building upon existing understanding and mental models. In order for learning to occur, a student must find a way to integrate the material that is presented with their existing mental model, or make changes to their model as needed to support the new information. As this is very much an intuitive process for instructors to participate in, as it requires empathy and understanding of a student's viewpoint, teaching and assessment cannot be completely automated in the best case. Human feedback is necessary to examine a student's submission and make the necessary comments required to adjust their understanding to be more in line with the accepted or correct model.

## 2.2 Traits of Good Grading

Before any discussion of grading tools can be undertaken, it is necessary to first establish some concensus on what the minimum requirements for "good" grading are. As has been pointed out within official UCR CS&E Department policies on instruction, there is a difference between assignments that are intended as practice and assignments that are intended as evaluation. The majority of student submissions are likely to be intended more as practice than as assessment. In order for practice to be as educationally effective as possible, these features are considered a bare minimum feature set for a good grading system[1]:

- *Response Time* - Clearly the speed with which feedback is generated is considered something of great value. One of the major features of computer-based assessment schemes utilizing computer-checkable discrete-response questions like MCQs, Matching, and short Fill-in-the-Blanks questions is the fact that students can instantly find out how well they are doing. Rapid feedback response is one of the most touted features of many on-line tutorials and quizzes [22] [11]. In the more general domain of human-checked work, having prompt feedback is a highly beneficial, if obvious, feature of a good system for CAA.

- *Accuracy* - Accuracy is another obvious requirement for good grading: if the scores have little or nothing to do with the submission then student motivation will undoubtedly drop.

- *Quality of feedback* - It is one thing for a student to receive an accurate score of their work shortly after submitting it, it is quite another to be given a detailed breakdown of what they did well, what needs work, and what didn't work at all. Quality of feedback

[1] This completely ignores the very important issue of what can be learned by the student in a given practice item, focusing only on what makes the grading of a given item more or less effective. This is by no means sufficient to ensure that a given assignment is pedagogically valuable to the students, but is intended a list of things necessary to maximize the pedagogical value of a given item.

is something that commonly slips through the cracks: Professors often look only for scores, while TAs and Graders are looking only to finish grading. The result is that it is not necessarily in anyone's immediate interests to ensure that the students are told what they did wrong. While there is some small feedback that students get from their numeric aggregate final score, it is not nearly so useful as if they were given detailed grading information.

- *Consistency* - Something that is time-consuming and difficult to achieve when grading without the aid of a grading tool is consistency from submission to submission. As any grader knows, the majority of errors on a given assignment are repeated by more than one student. In order to be completely fair to the students, each time a given error is found within an assignment, the penalty (and probably the feedback) ought to be the same. Without such consistency, student requests for regrades increase, student moral may drop, and complaints of favoritism may turn the group opinion of the students against the instructor. An extremely detailed rubric is the ideal method of ensuring consistency and stopping such problems before they start, but generating such a detailed rubric before knowing what errors the students actually made is a difficult art indeed. Failing that, a good system for grading should provide a method for boosting consistency, if not outright ensuring it, by assisting in the retroactive creation of a detailed rubric.

- *Flexibility* - The flexibility of a CAA system is an essential feature: how much does the system allow the grader to override? There is a facetious saying that says, "If you build a fool-proof system, they'll just build a better fool," which educators should be able to recognize as applying double in the face of student work. No matter how robustly a CAA system is built in terms of attempting to deal with the unexpected in student submissions, it is very important that the grader has ultimate control to override all of the system's actions. Otherwise, the grader will revert to the slower, but less restrictive,

method of grading by hand. If the flexiblity of a CAA system is insufficient, the other factors listed here will likely suffer.

# Chapter 3

# History of Computer-Aided Assessment

Computer-Aided Assessment has a history nearly as long as computing itself. The earliest documented reference to using computers in an attempt to simplify grading dates back to 1959 at Rochester Polytechnic Institute, where a computer program was used to test the behavior of student's machine-language submissions [17]. Work in the area has continued in an ad-hoc fashion over the ensuing 45 years, and can give us a general idea of the components that are regarded by the computer-science education community as necessary for a CAA platform to be useful. To date there is still a notable lack of a definitive tool for CAA that meets all of the technical requirements, has sufficient developer backing, and is intuitive, powerful, and usable enough to gain widespread acceptance. Sampling the developments that have been made and the projects that have been attempted over the last 45 years can give us a good map of exactly what is needed. In fact, several recent CAA papers specifically enumerate the design goals that their designers were striving for. It is upon the shoulders of these developments that Agar stands, and much of the discussion on the design of Agar in Chapter 4 is informed by these papers.

## 3.1 Forsythe and Wirth

The most commonly cited early paper on the subject of automated grading in the CS context is Forsythe and Wirth's "Automatic Grading Programs" [13] from Stanford. In this paper, published in 1965, it is mentioned that "grading programs have been used intermittently since 1961." The grading system that they present touches on many of the topics that have been brought up again and again in the ensuing years: tracking running time, storing student grades, terminating misbehaving submissions,[1] security, and non-binary "fuzzy" grading. These themes take on charmingly antiquated meaning given that security concerns for Forsythe and Wirth focused primarily on hoping that students did not alter the grader library code on tape, and that students would actually use the pre-printed cards to set-up and tear-down the grading framework within their own submissions.

One of the quotes from this paper that blesses the growing field of CAA is this, "We recommend grading programs to all who teach programming and numerical analysis to masses of students, but the prospective user should first carefully investigate the systems available to him."

In terms of categorization of the system presented in this paper, Forsythe and Wirth reduce all need for estimation about robustness and generality, "... it is relatively easy for us to write a separate grader for each problem, and furnish very detailed messages about each case."

## 3.2 Hollingsworth

Prior even to [13] was a paper by Jack Hollingsworth, published in 1960, describing activities dating back to September of 1959. This paper is noteworthy in a few ways:

[1] In this case, a mechanism for the computer operator to invoke the submission on the next test case

- Given 120 students in a full-semester programming course, “We could not accommodate such numbers without the use of the grader.”
- “Students seem to like the grader and are not reluctant to suggest improvements!”

Further, it is nice to see that the state-of-the-art has advanced some in the past 45 years: “The grader cannot be completely automatic. Over-flow, invalid addresses, built-in stops and other effects can make the computer stop, and will usually cause it to stop again during tracing. Manual reentry is necessary.” [2]

We also find that the claim is made once again that security is a concern: “Student programs can modify the grader itself.” Also, it is worth noting that a strong set of constraints were imposed upon the programs in order to make automated grading feasible.

On the other hand, this paper as well as [13] demonstrate strikingly the elegance forced upon programmers during those earlier periods of computing: both papers provide full algorithmic or code depictions of the grading systems. [13] provides full code for their grader in ALGOL, and [17] provides a description of the 108 machine-language instruction framework that made up his grader. For comparison, Agar is composed of approximately nine thousand lines of Python code.

This grading system can be viewed as a very primitive version of the Unix shell snippet

```
if [ ‘./studentProg < test1.in | diff - test1.out‘ ] ; then
  echo Incorrect
else
  echo Correct
fi
```

[2] It is nice that we are no longer talking about grading student’s programs on punch cards and requiring a physical reset upon error.

## 3.3 BAGS: Basser Automatic Grading Scheme

The Basser Automatic Grading Scheme (BAGS) developed at the University of Sydney in the mid 1960s represented a significant improvement in the state of CAA applied to programming exercises. For the first time, grading of assignments on a batch-processing system did not require special actions or knowledge on the part of the operator. The BAGS system was "simply part of the standard operating system and its exercises [were] run as normal, batch-processed jobs."[15]

In the paper describing the Bags system, J. B. Hext and J. W. Winings specifically enumerated their requirements for a CAA system:

1. It should handle exercises in ALGOL, in MINIGOL (a subset of ALGOL), and in the KDF 9 Assembly Code.

2. It should not place any additional burden on the operators.

3. It should record every attempt at an exercise, with sufficient data for calculating a mark.

4. It should provide summaries on request for specified classes and exercises over a given period.

[15] also gives concrete examples of the types of programming problems that were being graded in this context: well-specified mathematical operations with zero margin for error. The grading scheme used is well suited for this type of problem. However, in terms of modern Computer Science pedagogy, there is really no way to view these problems as anything but toy problems: easily solvable in less than 5 minutes and 20 lines of code in a modern programming language. While there was a relative flurry of papers published and work done on CAA approaches to programming assignments during the 1960s, the work is simply too far removed from the modern realities of computing to be anything other than a historical interest piece. To get a better notion of what CAA means, we must advance through the literature several decades.

## 3.4 Kassandra

In 1994, Urs von Matt from ETH Zürich published a paper describing Kassandra [24], an automatic grading system initially designed for Maple or Matlab code in a scientific computing course. Kassandra is implemented as a network service to which students may submit attempted solutions to programming problems to have them tested against a suite of test cases for each problem. Kassandra reports results back to the student. As it is designed to be publicly available throughout the term, rather than run by a grader after the due date, security becomes of even higher concern here. Malicious student programs should not be able to retrieve the reference solution from Kassandra's internals. To deal with this, and the prime reason for Kassandra being implemented as a network service, the system is split into client and server systems. Students submit their code to their local client, which is responsible for connecting to the server, thereby initiating tests on the student code and reporting results. At no time is the client code able to access the reference implementation, the stored test cases, or the graded results. While there are certainly some security concerns raised by introducing the added complexity of the network, it seems that this is an admirable solution to the problem of restricting access to the solutions.

In terms of actual grading, this is another instance of testing for precise output, but it also requires students to be explicitly aware of Kassandra by making (generally uninvasive) changes to their code. This is perfectly acceptable in the domain of scientific computing, much like compilers or in many cases operating systems. Specs can easily be made strict enough to require precise output for a given input. This does leave great difficulties for any assignments that are not strictly testing the functional behavior of an assignment, or for which such testing is difficult. While interesting as a good approach to dealing with the security problems inherent in a CAA framework, Kassandra doesn't necessarily give us any further insight into guidelines for the design of such a system.

## 3.5 Dalziel's List

In 2001 at the International Computer Assisted Assessment Conference, James Dalziel presented [12], which contains several enumerations of pedagogical and technological models describing CAA in distance-learning, particularly using the World Wide Web. Of particular note to this discussion is his listing of "the five criteria used to consider the usefulness of the systems":

1. *The ease of use*: Dalziel includes designers and students as users whose needs must be met and managed in this list, but others have also included instructors and graders as likely users.

2. *The technical proficiency required*: Again Dalziel focuses this in terms of the designer of a CAA tool, but for *any* CAA tool this is of great import, for exactly the same list of actors: designers, students, instructors, graders[3].

3. *The degree of special hardware or software required to incorporate the tool into learning*

4. *The ability of the tool to track performance data arising from use of the tool*: If we are talking about tools that are primarily focused on the grading of assignments rather than the administration of quizzes for assessment, this is perhaps less important, but remains a good goal in any case.

5. *The costs of using the tool to enhance learning*: These costs are most certainly measured not just in financial terms, but also in terms of the amount of learning that must take place for people to use the tool effectively in any of our given categories.

This list very concisely enumerates many of the important points that we learned during the development and deployment of Agar, as discussed in Chapters 4 and 5.

[3] If designer and instructor feels redundant, replace these terms with author and assigner.

## 3.6 Pardo

In [19], Abelardo Pardo provides a sample architecture for a large software system combining the requirements from Dalziel [12] with similar functionality similar to that provided by the digital drop-box features of Blackboard [8]. Pardo's system requires minimal technical knowledge on the part of instructors when setting up items of assessment; It provides a web interface for all configuration pertaining to a given assignment. The web interface is also used by students for submitting work, although the actual transfer mechanism used is in fact email. Submissions are tagged with XML information appropriate to the information being assessed and passed to various modules to handle automatic processing or human examination as appropriate.

The places where Pardo's system fails to meet Dalziel's requirements are in terms of usability for the most critical user of all: the grader. While students spend some minutes submitting assignments, and instructors may spend minutes or perhaps an hour setting up an assignment (actually performing the act of *assigning*), graders spend hours if not tens of hours dealing with each assignment, and this time is generally proportional to the number of students Pardo's huge system failed to handle the simple axiom of design: *Design for the common case.* Pardo's description of the grader interface is eight sentences long.

## 3.7 Fully Automated Assessment

At the Helsinki University of Technology, instructors facing larger and larger introductory class sizes have moved beyond simply having Computer-Aided Assessment into the realm of fully Automated Assessment (AA) [18]. Malmi, Korhonen, and Saikkonen acknowledge that fully automated assessment does not give students the depth of feedback that could be given if resources were available for more human graders and smaller class sizes, but at the point that courses have become too large to handle any other options, AA is far superior to

the other obvious alternative: self assessment or voluntary programming assignments.

AA has some benefits, some of which are appealing even when compared to traditional grading. The most obvious benefit is the reduction in resource requirements, as mentioned. Secondly, the system can give back feedback at any time of day or night. This second property leads naturally into the third: since feedback is returned very quickly and entirely objectively, it is sensible to allow students to resubmit based on that feedback. This is in line with the constructivist view of pedagogy that has taken hold in the educational community. If the constructivist view is correct, then there is certainly some value in AA. Whether or not there is value in human-checked assessement using the constructivist resubmission paradigm is not addressed in this work.

## 3.8 CourseMaster / Ceilidh

While [13] is the most commonly cited early work in the area of CAA, CourseMaster (formerly Ceilidh) is certainly the most commonly used and most commonly cited CAA tool for grading programming work. In [16], Higgins, Symeonidis, and Tsintsifas describe the grading system embodied in CourseMaster. The system that they present is both vast and yet narrow in design. It covers a large number of the functional requirements that have already been noted in this historical review, but there is no discussion of the human components that intuitively seem like an integral part of the evaluation and assessment process. CourseMaster appears to only support statically-checkable properties, and doesn't seem to explicitly have support for grader-override of marking. CourseMaster requires Java familiarity on the part of the grader. In these ways, while CourseMaster sounds like an amazing development, the published description implies that the system is forcing the user to conform to the system paradigm, rather than enhancing existing paradigms of usage.

# Chapter 4

# Design of Agar

## 4.1 Paradigms for CAA

There are a number of common and obvious paradigms for the development of CAA tools for grading programming exercises. Agar was developed with one of the more complicated of these approaches, but the whole spectrum of these paradigms are represented in Chapter 3.

### 4.1.1 Single-Purpose Testers: The "Simple" Approach

The "greedy" scheduling approach to Computer Assisted Grading is one that every programming-literate grader has probably dreamed of during the tedious hours of slogging through student programming submissions: "Why don't I just write a script that does steps $X$, $Y$, and $Z$ for me and show me the results?" Indeed, a well-written program spec for Computer Science classes can make the functional parts of grading almost completely automatic. This sort of grading aid, often written in a scripting language such as bash[14], perl[25], or python[20],

can reduce the time requirement for grading the functional behavior of a well-specified programming assignment. Unfortunately, this has minimal reusability and can do nothing to assist with having to manually inspect student work for non-objective requirements like style and documentation[1]

This approach, possibly in conjunction with a text file containing comments pertaining to common student mistakes, can be used with some efficiency. However, in order for there to be a net-reduction in time spent grading, the development of such a script must be relatively straightforward. For "meat-grinder" programs like those that are commonly assigned in lower-division CS courses (programs that take a well-specified input format and produce a completely deterministic output of an equally well-specified format), the majority of this task can be automated with two nested loops and proper invocations of the *diff* utility under Unix. Programs that have any form of freedom in interface are often ungradeble using this approach, as are many upper-division programs where the complexity of the average assignment prevents any simple method for automating the testing. In general, this approach can pay off much better for lower division lecture courses with large enrollments. The development time is generally not feasible for courses with difficult-to-test assignments, flexible requirements, or low enrollment.

### 4.1.2 Reusable Code

After a number of attempts at using one-shot scripts for grading, the computer scientist will begin to mull over methods of designing a more robust system requiring a shorter development and testing cycle per assignment. In the author's experience, the most obvious steps in grading are those of identifying what files are actually a useful part of a student submission, compiling the student's submission, and writing out the testing report for the student. Since all or nearly all programming submissions will involve these steps, these are the most likely

[1]The possibility of using a static analyzer like lint[6] to enforce certain style requirements is also known.

first targets for a more reusable framework for CAA.

If these tasks are automated in terms of a library to be incorporated into a larger, single-use program for grading an assignment, flexibility and transparency are kept very high, but the efficiency of this approach is questionable

### 4.1.3 Agar: A "Generalized" Framework

Agar[2] was developed with the intent to be as general as possible, with the idea that if the functional tests that are provided with Agar are found to be insufficient, it should be easy for a grader or instructor to develop new tests for the assignment in question with relative ease. The details of this interface are presented in Sections 5.2.1.

Additionally, Agar represents a significant advancement over previous attempts in that it also has shown greatly reduced effort required to grade *non*-programming material such as written work and quizzes, and can be incorporated with an Optical Mark Recognition system that was also developed at UCR. The automated-testing features of Agar make it ideal for grading programming work of all kinds, but the real benefit comes from the time-savings for human graders in providing detailed feedback. Additionally, Agar is open source and is written in an interpreted language, so anything that isn't handled by the tool interface can be added to the system internally.

Agar addresses consistency and grader efficiency together. The first time a human grader using Agar finds a problem with a student submission that wasn't automatically detected (or even tested for), they create a new comment, assign a point value (positive for bonus, zero to just write a comment, and negative for penalty) to the comment, and write out a note to the student. A drag-and-drop system within the Agar interface then allows that comment to be assigned to any other student that is found to have the same mistake. Further, since

[2] Agar is not an acronym, nor does it have any intrinsic meaning. It was chosen because it vaguely conjures "automated grading", but doesn't violate the author's distaste for acronyms.

comments are assigned by reference, the point value or feedback can be changed later on, and all submissions that received that comment will automatically be updated.

This comment system allows for much greater feedback to be generated for each student in a much shorter amount of time. For C++ homework in our lower-division courses, a two-week project for a class of 60-70 students now takes about 4-6 hours to grade, to record scores, and to generate and send out detailed feedback for students. Previously, lower quality feedback and less accurate grading took at least 10-15 hours. Similarly, 12 written problems for the same course were graded and commented in just under 5 hours, or slightly less than 5 minutes per student. Students in this course have how helpful they find it to get their work returned to them within 1-2 days of turning it in, and have detailed comments and feedback emailed to them while they still remember what the assignment involved. Graders are similarly pleased in that they get to do more in less time, and no longer have any bookkeeping to do since Agar can automatically exports its results to a course grade-book in the form of a spreadsheet.

## 4.2 Initial Design Decisions

Usability was a major concern in Agar's design from the very beginning. Since the initial aim was to reduce the amount of time it took a grader to grade programming, an elegant interface was regarded as essential. The general task of grading **must** be easier to perform using the electronic system than when performed by hand, otherwise there is no reason to use the system. Ideally there will be a one-to-one mapping of tasks in the traditional paradigm to tasks using the tool, and each of those tasks should also see some sort of increase in efficiency. The "clerical" tasks involved in grading should be automated to the greatest extent possible, meaning that grading information should be exportable in a number of useful formats, with as detailed information as possible, and hopefully strong integration with any courseware

systems in common use. Security is of prime importance, because the act of running a student's submission is very much an act of running untrusted code[3].

One of the most fundamental design decisions, and one that has stuck throughout the development of Agar and into the planning stages for Agar2, is the idea that all automated decisions must be over-rideable by the human grader. Any system that is designed with the (probably false) assumption that all error conditions have been accounted for is going to be difficult to work with, and it is the pinnacle of hubris to think that this or any other CAA tool has been so well designed as to not need[4] human oversight.

Another powerful and common idea is that the tool should require no alterations to student's programming behavior. It should, like all good systems, "just work". Some changes to assignment specification (i.e. more accurately defined specifications) are allowable, but Agar should be minimally invasive for both the students and the designer of an assignment. Similarly, the ability to work with assignments regardless of programming language is important[5]

In the inevitable event of a grading dispute, the system should leave a sufficiently strong paper trail that even if the original grader should meet an unfortunate accident, there should be no risk of unnecessary disputes.

Finally, it was initially thought that the system should be relatively extensible. A simple interface was designed such that programs could be written to extend the tool set whenever the existing tools didn't meet their needs. A fair amount of flexibility and efficiency was sacrificed in order to make this interface as simple as possible[6]. An interesting, and in hindsight predictable, note is that this extensibility has never been taken advantage of. Only a

[3]In general this will not be intentionally malicious code, but students are notoriously capable of discovering new and violent ways of making their programs crash.

[4]Or in most cases, allow.

[5]In the CS&E department here, we have assignments in C, C++, Python, VHDL, an assembly language, and Perl on a regular basis. Ideally these should all be equally supported.

[6]Details of the tool interface can be found in Section 5.2.1

single user during the entire deployment lifespan has expressed any interest in utilizing the tool interface, but lost interest when it was explained that some of the existing tools could be leveraged to accomplish the desired testing. Agar2 will certainly take advantage of this insight.

## 4.3 First Quarter Changes

Initially Agar, like many of the CAA systems discussed in 3, tried to enforce only binary grading decisions in the automated testing process. It was hoped initially that Agar's automated testing framework could identify those things that didn't require any human intervention (those that met the spec perfectly), and initially give no credit for any test that did not have a positive outcome. During the first quarter of deployment, this was quickly identified as one of the worst assumptions. Simple things like docking a number of points per day late could not be easily expressed with such a system, and so support for non-binary grading was introduced. In light of constructivism in Computer Science pedagogy, this seems quite obvious. It is far better to support methods that can encourage additional intuitive leaps than it is to immediately punish any understanding that is not wholly correct.

## 4.4 Second Quarter Changes

A concept that was finally made brutally clear during the second quarter of deployment[7] is the idea that whenever possible the tool should enhance and enable existing usage. A primary example of this are the ideas of a spreadsheet and email. Both can used in ways that are very similar to existing usage paradigms. A spreadsheet can be used for double-

[7] An earlier understanding of this idea would have greatly simplified the deployment and development process.

entry bookkeeping in a way that is almost exactly the same as bookkeepers would do by hand, but enables greater usability through automation. Email, similarly, works under exactly the same paradigms as mail service in the physical world[8], but enables much more rapid delivery, multiple recipients, and other features. This notion of being an enabling technology under existing paradigms is powerful, since it allows users to leverage existing models of understanding when confronting the program for the first time. Forcing a user to not only learn a new tool but also an entirely foreign paradigm for completing a task will cause undue stress on the user and great resistance to adoption.

A related design point that was only made clear after many cycles of development and deployment is the fact that a system like Agar has little similarity to any common programs. It is not a browser, a shell, a word processor, or a spreadsheet. The users of the system will simply have no initial mental model of the system, so if the program model is not obvious, the discrepancy between these models will cause some stress on the user and will become a great barrier to adoption[9].

[8]Ignoring setting up mail service settings, which is a relatively simple one-time cost.

[9]In short, the learning curve can be insurmountable if the workings of the program are not completely transparent. This has been a problem in the development of Agar, but is one of the prime things that we hope to fix in Agar2

# Chapter 5

# Technical Details

## 5.1 Views

In [12], James Dalziel discusses four primary user types or roles into a CAA system: student, designer/author, instructor, grader. Agar explicitly alters only a single one of these, the grader role. The student and instructor roles are handled here at UCR CS&E by the *Turnin* system, which provides a web-interface for instructors or TAs to create electronic drop-boxes for assignments. The students use a similar web-interface to select a course and assignment to submit into, are notified whether their submission was on-time or late after submitting, and are given a cryptographically signed receipt for their files. Agar has been developed to work primarily with the directory of student submissions that Turnin generates, but the two systems are not yet tightly coupled in any way.

In general, the use of Agar can be completely hidden from the students. An intriguing and so-far unused possibility exists to generate a rubric under Agar bundled with sample input and output files and distribute this to the students. By doing so, the student can run the same automated tests on their own code before submitting, thus reducing any chance of

misunderstanding in the problem statement and shifting some of the burden from the grader back to the student.

Agar may slightly affect the role of the designer or author of an assignment, by altering the types of things that are assigned to be easier to grade within the Agar framework. Programs that have well-specified input and output formats are very well supported by Agar, as are programs that can be graded by running regular expressions against the source, but complicated programs that have a great deal of imprecision or creativity leverage less from Agar. This is really a secondary effect on behavior, rather than a required behavioral change, but is worth mentioning.

## 5.2 Details of Agar

At the highest level, grading using Agar consists of two primary tasks: creation of a rubric and associated automated testing, and then manual inspection of individual submissions where needed and the distribution of Comments to override or augment the automated scores. These two tasks are fairly well distinguished, and have been separated within the Agar interface as demonstrated in figures 5.1 and 5.2.

Developing a rubric is primarily a matter of dragging tools from the Toolbox (the left-hand pane of Figure 5.1) into the tree structure of the Rubric (the right-hand pane of figure 5.1). It is generally also useful to identify which files are part of a student's submission at this point, so as to be able to more accurately test the rubric to ensure maximum benefit from the automated testing. The "Submission Manager", discussed more fully in Section 5.2.2, is responsible for this task.

The Grading view is used to manually inspect submissions. The files corresponding to each submission are displayed in the tree structure left-most in the Grading view. In Figure 5.2 the usernames of students have been replaced with anonymous grader tokens, as this set of

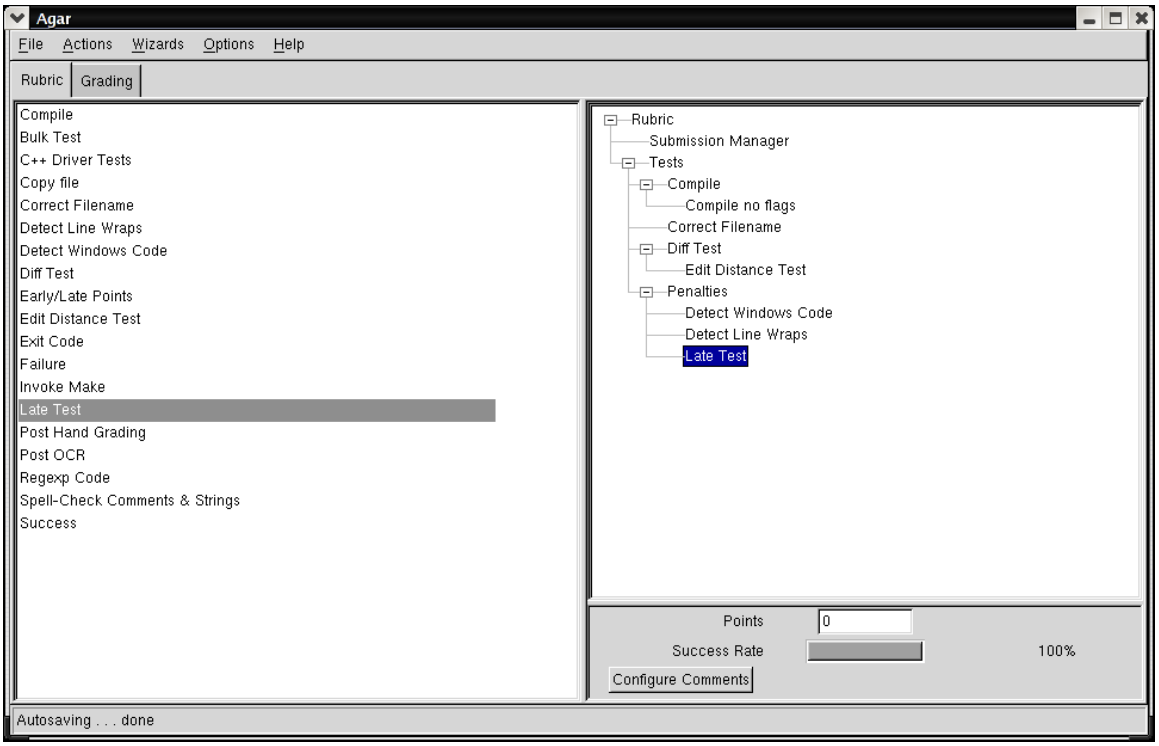


Figure 5.1: The Rubric View

submissions is an artificial test. In normal grading the username of the student corresponding to each submission would be displayed here. The center portion is a list of Comments, which are small objects that have a point value (positive, negative, or zero), an associated top-level rubric item, and some associated text. Drag and drop from the comment list into the right-most pane (the submission window) will add the given comment to the currently selected submission.

Once all of the automated tests have been executed, and all necessary human evaluation has been performed, mailbacks for each student can be sent out individually from within the Agar interface, and detailed results can be exported to a spreadsheet. Some individual features of this process deserve some attention, and are described below.

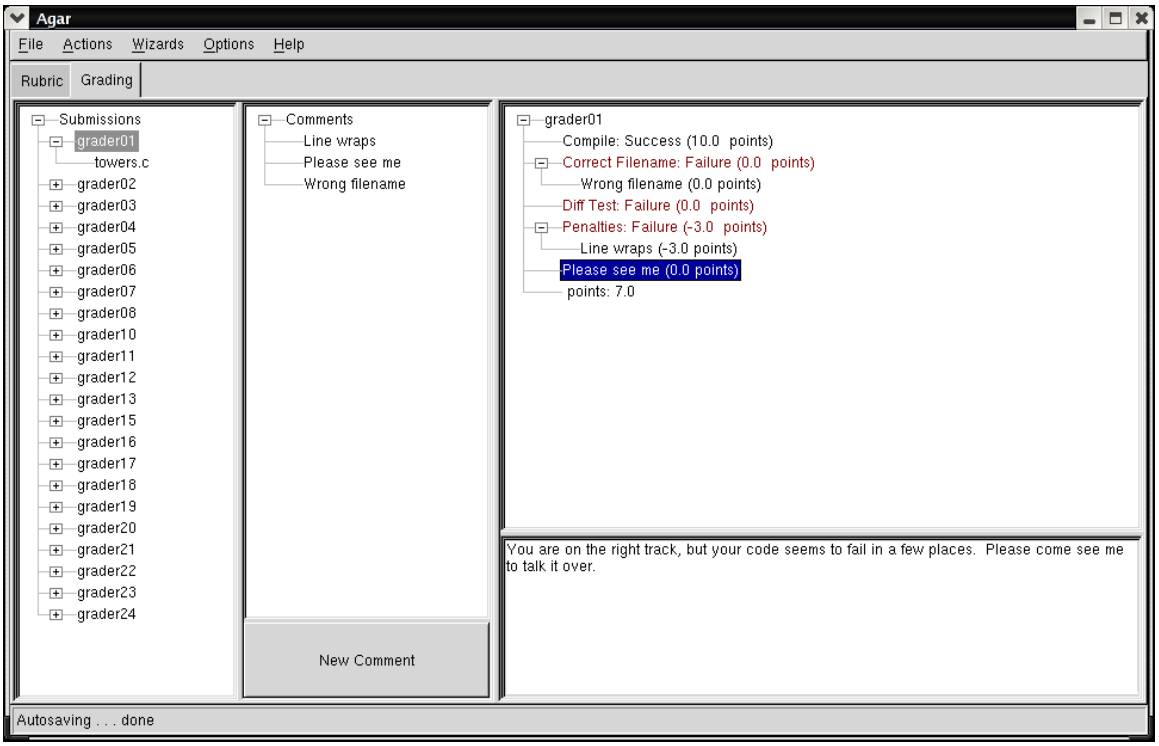


Figure 5.2: The Grading View

### 5.2.1 Tools

The core idea behind the automated testing in Agar is to compose a rubric out of extensible, reusable, hierarchical tests. A rubric is composed of a number of “top-level” rubric items which are displayed in student mailbacks and detailed grade records. Each of these top-level items may have sub-tests that are executed in the case of a failure.

As mentioned in Chapter 4, in order to ensure maximal flexibility in testing tools, the interface for tools in Agar was intentionally kept as simple as possible. For the majority of tools this works quite well: each tool can be written to test one submission at a time and return a simple Pass or Fail value. Agar itself can then determine appropriate point values as a result. However, some tests are non-binary. One of the first places this became a difficulty was in the grading of multiple choice questions for an upper division course wherein the instructor desired a “guessing penalty” to be applied to any wrong answers.

Thus each question had 3 possible outcomes: a positive score for a correct answer, a zero for no answer (an admitted "I don't know"), and a negative score of an incorrect answer. This could have been dealt with by writing two separate tools for the testing: one that checks for a correct answer, and a secondary that is invoked when the first fails that checks for an incorrect answer. However, even at the time that this was first becoming a known issue the author was becoming increasingly aware that some grading is simply not a binary operation[1]. A more "advanced" tool interface was thus developed where testing tools can choose themselves how many points are to be awarded for a given invocation of the tool.

The choice was made to extend the original tool interface rather than completely redevelop with a more complicated (and rigorous) system for extending the toolset. It is very much not in line with the spirit of the project to force users to spend hours developing custom tools when a less robust interface would allow for faster development in the common case. As noted in Chapter 4, this decision turned out to be rather unimportant, but very much affected the complexity of the tool interface.

### Tool Interface

The Tool interface, rather than relying on shared libraries or intricate IPC like most plug-in architectures [23] [3], was designed so as to be easily explained in a couple paragraphs worth of text. No complicated headers are needed, no language requirements are imposed. In fact, it is difficult to imagine a simpler extensibility interface that still runs extensions in a separate process space.

The full requirements for the original tool interface are as follows:

1. Each tool must be executable in the standard UNIX syntax for executables. This im-

[1]This decision was strikingly upheld two months later when it was discovered that there was need for a "Bulk Testing" tool that would test a submission against a number of outputs at once and return what fraction of those were a match. To do this using a purely binary system would require, in this case, a 30-fold complexity increase, which could cause the automated testing to take longer than a human doing the same manually.

```
Usage: difftest.py [OPTION]... EXECUTABLE
Return exit code 0 or 1, 1 indicating output differences
0 indicating no differences.

--infile=FILE           What input should be given to the executable.
--outfile=FILE          What output the executable must produce on
                        that input.
--timeout=secs          How many seconds before giving up.
--as-argument           Give the program the argument 'infile' rather
                        than sending the contents of infile to standard
                        input.
--flags-to-diff=FLAGS   Optional flags to be passed to diff.  See
                        the man page for diff for more info.
```

Figure 5.3: Usage Message for difftest.py

plies that the execute bit is set on the file, and the file is either a binary executable or an interpreted script with the proper shebang[2] header line.

2. When invoked with the –name option, the tool must print to standard output a long-form name for itself and then exit with return code 2.

3. When invoked with the –help option, the tool must print to standard output a usage message with long-form (GNU-style) arguments. Flags should be specified without an "=" character. File arguments should be specified as "–argument=FILE". All other arguments should have an "=" character and some value other than "FILE".

   For example, Figure 5.3 shows the usage message for the *difftest.py* tool. This message will be parsed by Agar, and the configuration dialog shown in Figure 5.4 will be generated, thus obviating any need for the tool developer to have any knowledge of GUI programming, Python, or the wxPython widget set.

4. Each tool operates on a single file per invocation. That filename will be passed to the tool as the final command line argument.

[2] For example, #!/usr/bin/python

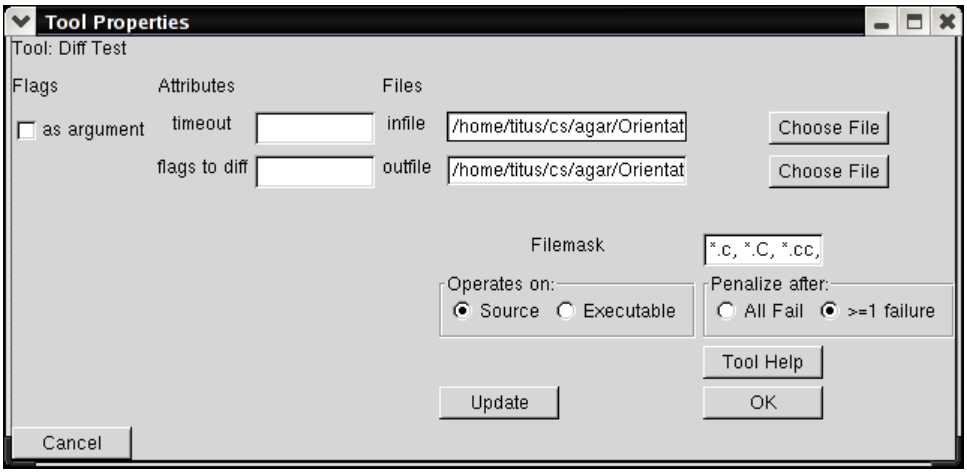


Figure 5.4: A Dynamically Generated Tool Configuration Dialog

5. For success, the tool should return an exit code of 0. For failure, the tool should print relevant error messages to standard output and return an exit code of 1.

When it was realized that binary grading was insufficient, an addendum to the Tool interface was made:

1. Upon termination, a tool wanting to return a non-binary value must output as its first line of output the number of points to be awarded. Only a single decimal place is considered valid (i.e. 10.1 is valid but 10.15 will be truncated).

2. Since the tool is being granted total control over the points awarded, most of the standard tools that utilize this interface also take point values as arguments. For example, the *pointlate.py* tool, which grants increasing bonus points for submissions up to 3 days early and increasing penalty points for submissions up to 3 days late, has the interface seen in figure 2.

3. Upon termination, a tool using this “advanced” interface must return exit code 3.

Every tool in Agar uses only this interface, with the exception of the compile tool. The compile tool (see Figure 5.2.1) has slightly more complicated behavior in that it needs to be

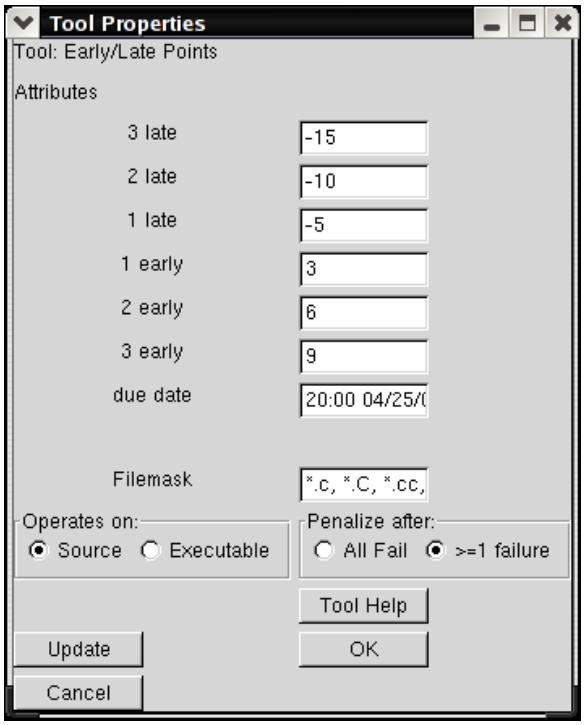


Figure 5.5: The Early/Late Points Configuration, Demonstrating Common Usage for the Advanced Interface

able to register an executable for a given submission in the event of a successful compilation. To date, there has been no need for any other special cases, although for the sake of efficiency the “Success” and “Failure” tests have been hard-coded into the system rather than invoking the overhead of spawning a child process that exits with a given return code.

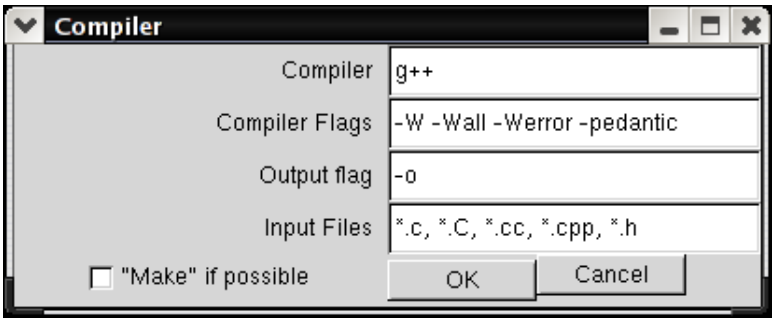


Figure 5.6: Compilation Dialog

### Existing Tools

Early in the deployment process there were frequent requests for new tools or additional features for existing tools. During the most recent quarter of deployment, only a single request of this type was submitted, which indicates either of two things. The first possibility is that the tools that are currently part of the standard toolbox are now sufficient that most common cases are well handled. The second is that the (now regular) users of Agar have adapted assignment specifications and grading styles to what Agar most easily provides, reducing the need for additional tools. In either case, the existing toolbox is fairly robust, and is presented here in Appendix A.

### Hierarchical Testing

One simple but novel feature of Agar that has not been explicitly described in other CAA systems is the notion of hierarchical testing: if a given test fails, run one or more secondary tests and return the sum of their outputs instead. This process can proceed recursively for several levels, although there has been no use for anything beyond two levels so far. A very common usage pattern for hierarchical testing is for compilation: In our lower division courses at UCR, student submissions have been required to compile with the *gcc* flags *-W -Wall -Werror -pedantic*, which ensures total ANSI spec compatibility and that the majority of common warnings will halt compilation. Since it is generally considered unduly harsh to not accept submissions simply on the basis of having warnings in the code, if a submission fails to compile with these flags, we may allow a secondary test (worth fewer points) that invokes the student's makefile, and a tertiary test that manually specifies compilation with g++ and no warning flags. Since compilation represents a significant portion of the computational cost for grading an average assignment, reducing the number of required compilation attempts in this way can make things much easier, and make the top-level rubric much cleaner.

Visually, due to the limits of the wxPython widget paradigm, an intuitive method of displaying this “test-on-failure” behavior has yet to be developed. Figure 5.2.1 shows the GUI representation of the compilation hierarchy described above.

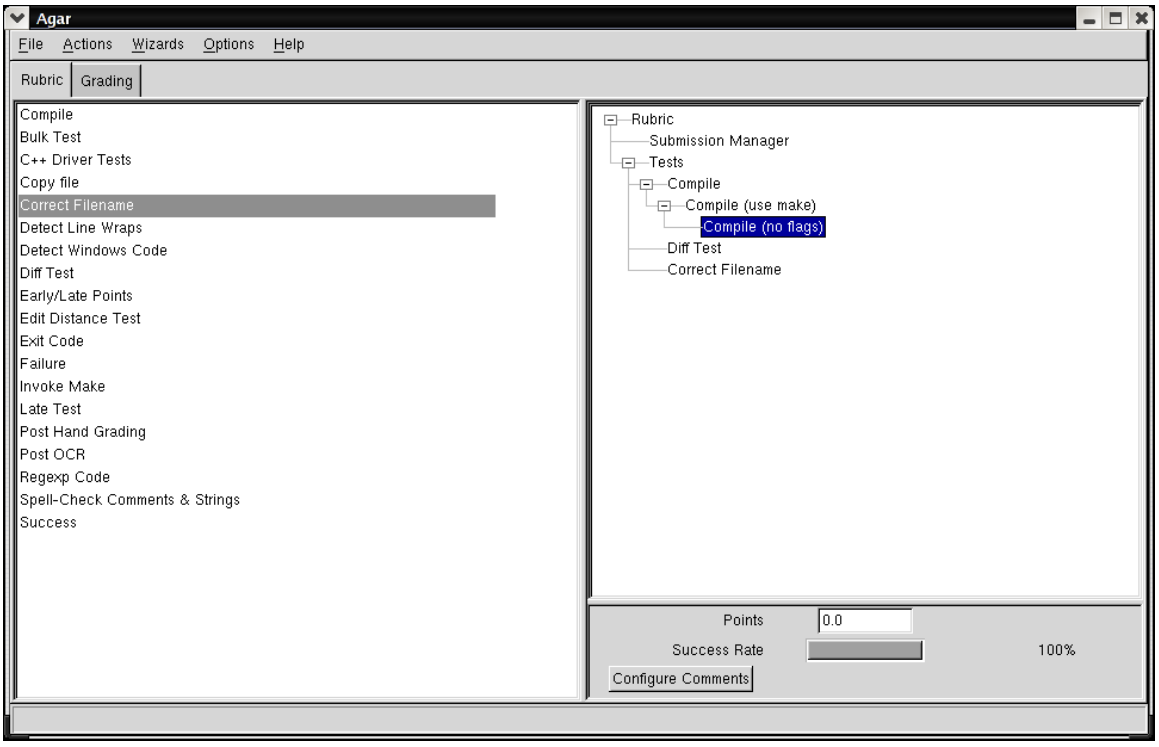


Figure 5.7: Hierarchical Testing

### File-Masks & Testing

One of the most important, and unfortunately subtle, concepts in Agar is that of the file-mask. Every rubric test has an associated “file-mask” field. When the tests are run, each file of each submission is compared against the file-mask for each test[3]. Any files that match are given to an invocation of the test. Depending on the settings for an individual test, points will may awarded if any invocation of a test for a submission passes or only if all invocations of a test

[3]In pseudo-code terms: for each test: for each submission: for each file: if the file matches the file-mask, invoke the test on that file.

for a submission are successful. If no files in a submission match the file-mask for a specific test, then no points are awarded.

Agar supports the concept of a "Default File-Mask", which is automatically set on startup if the Startup Wizard is utilized. For example, if the "C/C++ Code" option is utilized, the default file-mask is set to "*.c, *.cc, *.cpp, *.C, *.h". Any tool whose file-mask is not explicitly changed will utilize the default. This has the difficult side-effect of making the file-mask nearly transparent to the user most of the time. However, when the file-mask is wrong, users have a tendency to not understand what is happening, and get frustrated. Agar2 will have much more explicit and transparent file filters.

### 5.2.2 Submissions & Comments

The two other primary data structures in Agar are Submissions and Comments. Submissions contain a username, a list of files, test results, and a list of Comments. Comments pertain to a top-level Rubric item, or are listed as a non-specific "General Note." Comments also include a point value (positive, negative, or zero), and an associated textual note. Since Python deals primarily in references-to-objects, when a comment is assigned to multiple Submissions, any change to the base comment is reflected in all assignments of that comment. This increases the fairness of the grading immensely: it not longer requires any thought or effort on the part of the grader beyond remembering which comment corresponds to which problem (succinct naming is necessary).

### 5.2.3 A Grading Example

The overall usage of Agar can be demonstrated nicely with a concrete example. To begin with, we assume that student submissions are organized in a directory structure identical to that provided by the CS&E department's Turnin facility: directories look like *user-*

*name/ONTIME/* or *username/LATE/* and contain all of the files for a student's submission. Other directory structures are easily supported using the Submission Manager, Agar merely defaults to this structure.

Initial startup of Agar is easy, simply run the *agar.py* script. If invoked with no arguments, it will bring up a dialog prompting you to choose "New Workspace", "Load Workspace", or "Quit." The usage of these should be self-evident, shown in Figure 5.8 Beginning a new

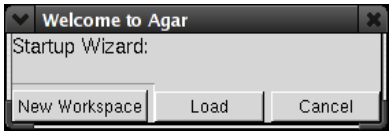


Figure 5.8: Startup Wizard

grading project requires us to select "New Workspace." This action brings up a secondary dialog, asking what type of assignment we are grading: Written work (wherein Agar is used primarily for the Comment facilities and grade reporting), C/C++, or Other (which prompts for an appropriate default file mask)[4]. The final step in setup is to choose a base submission directory, the directory that contains all of the student submissions. For example, if our student submissions are contained in */home/titus/grading/as1*, then we need to select that directory as the base submission directory.

Once the type of submission has been identified, Agar will automatically invoke the Submission Manager on the base submission directory and identify those submissions that it can. If there are directories contained in the base submission directory that do not contain any files matching the default file mask, then a warning is raised indicating there may be problems in identifying submissions and the Submission Manager may be invoked by the grader.

At this point, the submissions should have been identified, and the act of constructing a rubric may begin. Tools are selected from the Toolbox (recall Figure 5.2) and are drag-

[4]A fourth option, pictured, has been deprecated since the Marksense system has been moved from OCR to Optical Mark Recognition (OMR), and has been split out of Agar into an entirely command-line based suite of tools.

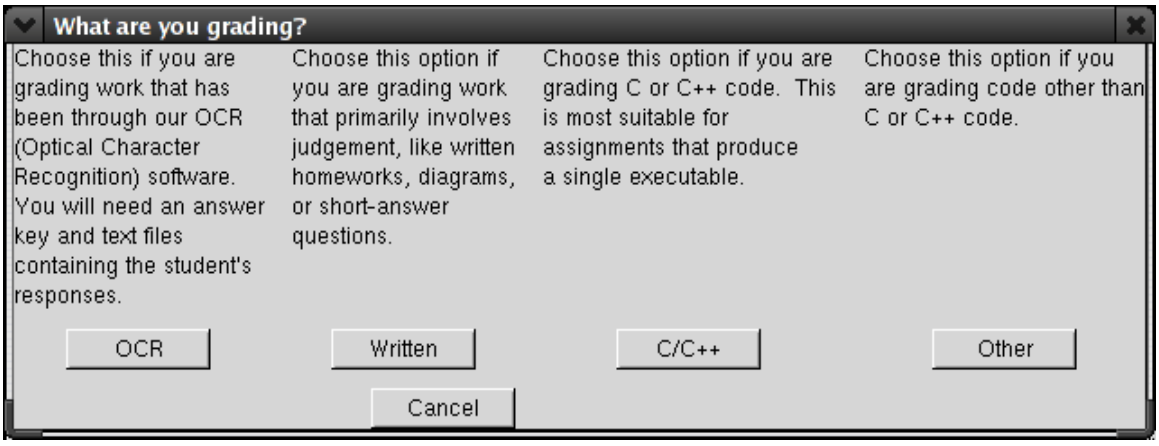


Figure 5.9: What Type of Grading?

and-dropped into the Rubric. Items in the Rubric may be re-ordered with drag-and-drop as necessary. Each rubric item may be configured by double-clicking, specifying options for that tool, changes to the filemask, etc. Point values for each rubric item are assigned by selecting the item in the Rubric and changing the text field at the bottom of the window. Note that each test can only be used to grant points (submissions start with zero points and test results increase this), as it was generally felt that Agar's rubrics should as closely correspond to an educationally accepted rubric as possible. To assign a penalty, an auto-commenting system is employed: individual tests can be configured to automatically assign a comment when a submission fails the test, or when a submission passes the test. Since comments can assign negative point values, this is the preferred method for assigning penalties. The Auto-Comment dialog interface is quite simple, and is shown in Figure 5.11

Once the Rubric is configured, the tests can be run. A progress dialog will be displayed showing the percentage of tests that have been executed and the username currently being checked, and any manual examination necessary can be done on the "Grading" tab.

From within the Grading tab, the grader has instant access to all of the files for each student's submission. Double clicking a file or a submission launches a viewer appropriate for the files in that submission. Upon inspection, if a submission is found to be lacking in

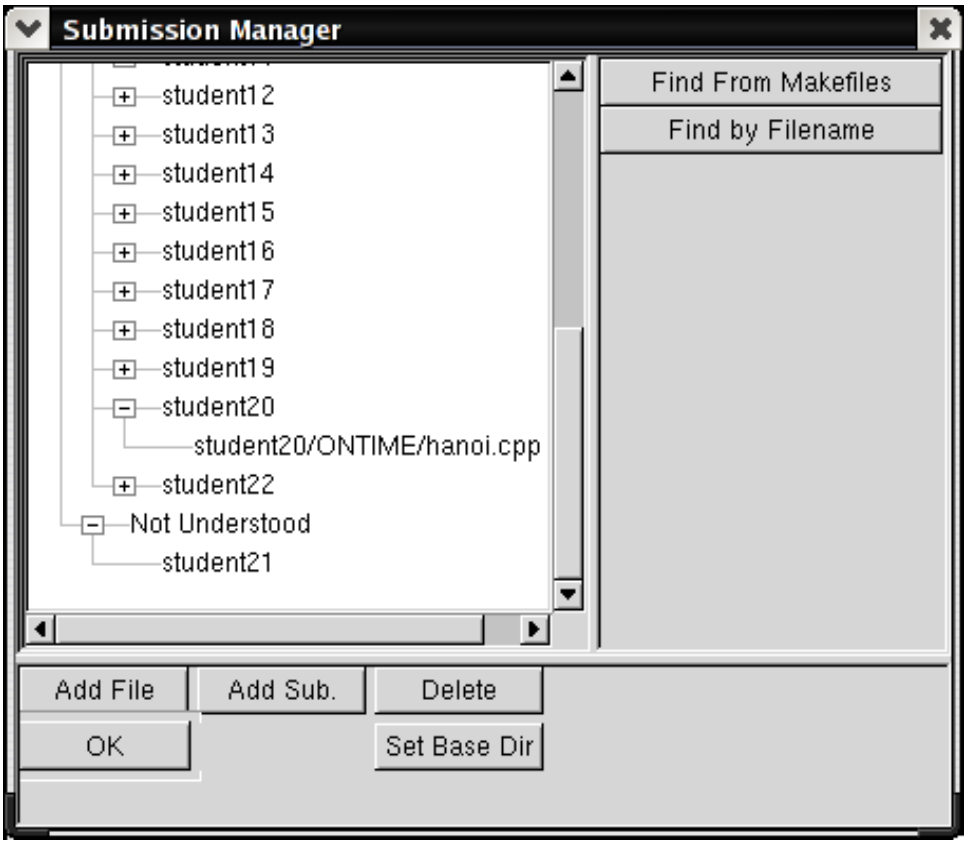


Figure 5.10: Submission Manager

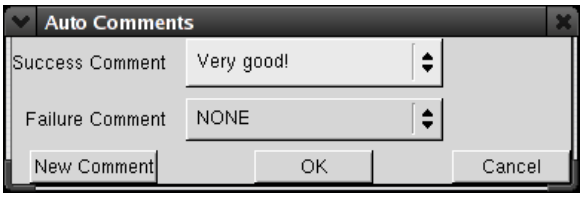


Figure 5.11: Auto-Comment Dialog

some respect, a new Comment can be created, assigned a relevant top-level rubric item, a point value, and a description, as shown in Figure 5.12. By default, creating a new comment assigns it to the current submission. At any point in the future, if the same problem is seen when the grader is examining another submission, drag-and-drop from the comment list to the submission pane will add the comment to the new submission. If for some reason the grader wants to work with the submission files, the context menu for each submission has

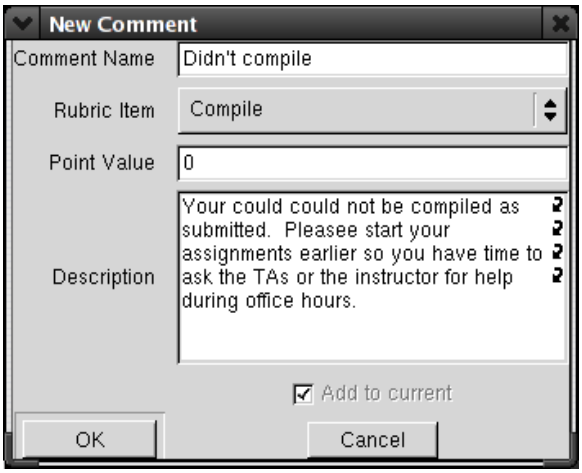


Figure 5.12: New Comment Dialog

an option to bring up a new terminal in the submission directory. There is also an option to re-run the rubric tests on a single submission. The majority of time spent grading programming assignments is spent performing these tasks: examining submission files, creating and assigning comments, working with submissions to see how close they were to working properly, re-running tests.

When the examination of submissions is complete, individual summaries of scores and comments can be generated to text files and mailed back to the students. Students have consistently commented on how much they appreciate getting detailed feedback, and using Agar it is not particularly more difficult to give detailed feedback than it is to give no feedback at all. Of great note to the instructor is the fact that when exporting grading results to a spreadsheet, scores are not merely aggregated into a single final tally per submission. Rather, a new sheet is created that contains a column for each top-level rubric item and the score for each submission on each item. This is particularly useful in conjunction with the OMR software for checking whether the answer key was entered correctly. In such a situation, if most of the students get a particular question wrong (which can be seen in most spreadsheets by simply highlighting the column corresponding to the question), then the key can be checked to make sure the students were graded correctly. Eventually, this level of detailed output will be

used in the CS&E department's ABET accreditation activities, allowing us to data mine the fine-level data and extract what students really know.

# Chapter 6

# Development Methodology

One of the unforeseen areas where Agar is of some interest is in terms of base development methodology. In an attempt to maximize user feedback in terms of bug-reports and interface design, Agar has been regularly deployed on the UCR CS department servers since September of 2003, and pushed as a useful tool to the TAs regularly during the ensuing period. All told, Agar has gone through no fewer than 15 iterations of deployment (on average, a new release at least every other week). In essence, we are using our local collection of users (drawn from the instructors and TAs) as testers throughout the process.

A number of factors contributed to such a rapid development cycle being possible. The most major of these being the choice of development platform. Agar is developed using wxPython [4], the wxWidgets [5] bindings for the Python [20] programming language. One of the major benefits of using Python is its very loose run-time type system, which allows for very rapid development and tactical changes. wxPython adds powerful GUI support in this same vein, granting simple development of advanced features. For example, the original addition of Drag-and-Drop support for moving Tools from the Toolbox into the Rubric took on the order of 20 lines of code for the base functionality. More traditional development platforms (C++, Java) could have easily had 7,000 lines (the size of the base project, not

counting Tools) of GUI code, to say nothing of the functionality that is required to make such a large system work. Just as Agar is intended to be an enabling technology for graders, wxPython is an enabling technology for building GUI applications of this sort.

Additionally, a number of factors contributed to this style of development being *necessary*. First and foremost was the fact that nobody could enumerate the majority of use cases for this project before development began. Certain features, such as a simple Tool interface and the "write-once" Comment system were known before beginning, and it was known that some form of submission identification system would be needed, but the majority of the Tools themselves, as well as the usage of Agar for grading written work, could not have been predicted *a priori*.

In essence, development on Agar was a variant on the Extreme Programming (XP) [1] methodology in terms of planning, deployment, and redesign. Release cycles were very short. Features were only attempted when it was felt they were understood enough to make a complete implementation of them before the next release[1]. Only a single developer was working on Agar for the majority of its lifespan to date, so simplicity was highlighted: it is far better to have something that works for most cases and satisfy the user than to delay the release in order to deal with esoteric situations that will likely never arise. This was not, however, full XP development. Rather than having a single client, development responded to the needs and whims of all of Agar's users at once. Rather than have a well-designed testing plan throughout, features were added in the most easily available fashion possible, which was later discovered to violate the layering of interface from functionality, making XP-style automated testing intractable.

[1]Note that the CVS repository never branches

# Chapter 7

# Future Work

As Agar approaches stability and a feature-set robust enough to stop the flow of requests, the places where it falls short are becoming more and more apparent. Some of the problems that are surfacing are subtle effects of the choice of wxPython as the GUI platform for the tool. Most notably, Agar fails to perform in one of our design guidelines: since this represents an entirely new kind of tool there is almost certainly going to be a large discrepancy between the user-model and program-model of behavior. To minimize this effect, the behavior of the system must be as transparent as possible. Unfortunately, within the wxPython widget paradigm, there is little that can be done to mitigate this elegantly. A fundamentally different interface design is necessary. If the interface is to be thrown out, it represents an opportunity for change in terms of separation of interface from functionality, increased testability, and better software engineering practices. Within the next 6 months we hope to begin work on Agar2, which will be a significant departure from the existing paradigm in terms of interface and usage metaphor, and will take advantage of all of the lessons that have been learned in the development and deployment of Agar over the past year. The core ideas in Agar2 will include

- A more concrete program metaphor. Agar meets Bruce “Tog” Tognazzini’s definition of a Graphical interface rather well [21]. Much better would be meeting Tog’s guidelines for a “Visual” interface: an interface where the application has a clear metaphor which is clearly and consistently communicated with the user through multiple channels (text, graphics, etc).

- A more robust system for combining tools into rubric tests, even at the expense of the simple tool interface. Users do not want to engage in tool development as a first step in grading. When faced with that, they will (historically) choose to grade manually even if the net time spent on the task is possibly greater.

- A more transparent testing phase. Currently Agar gives no more information than displaying the username of the submission being tested. Showing the exact flow of information and series of tests that are triggered during an execution would decrease the time required to debug a rubric.

- Tool “mix-ins”: So many tools have demonstrated a need for similar functionality (such as a timeout mechanism) that it seems prudent to allow for functionality mix-ins to reduce code replication.

- A proper separation of GUI and functional elements. One of the properties that was lost in the organic development of Agar is the separation of functionality from interface elements. Without this separation, regression testing is almost impossible. With it, the interface becomes just another modular component, allowing for greater flexibility in interface.

As a proposed system model, a flow chart with labeled and color-coded inputs and outputs would grant both greater flexibility and more intuitive interface. A grossly simplified mock-up of what a rubric could look like with such a system is shown in Figure 7.1.

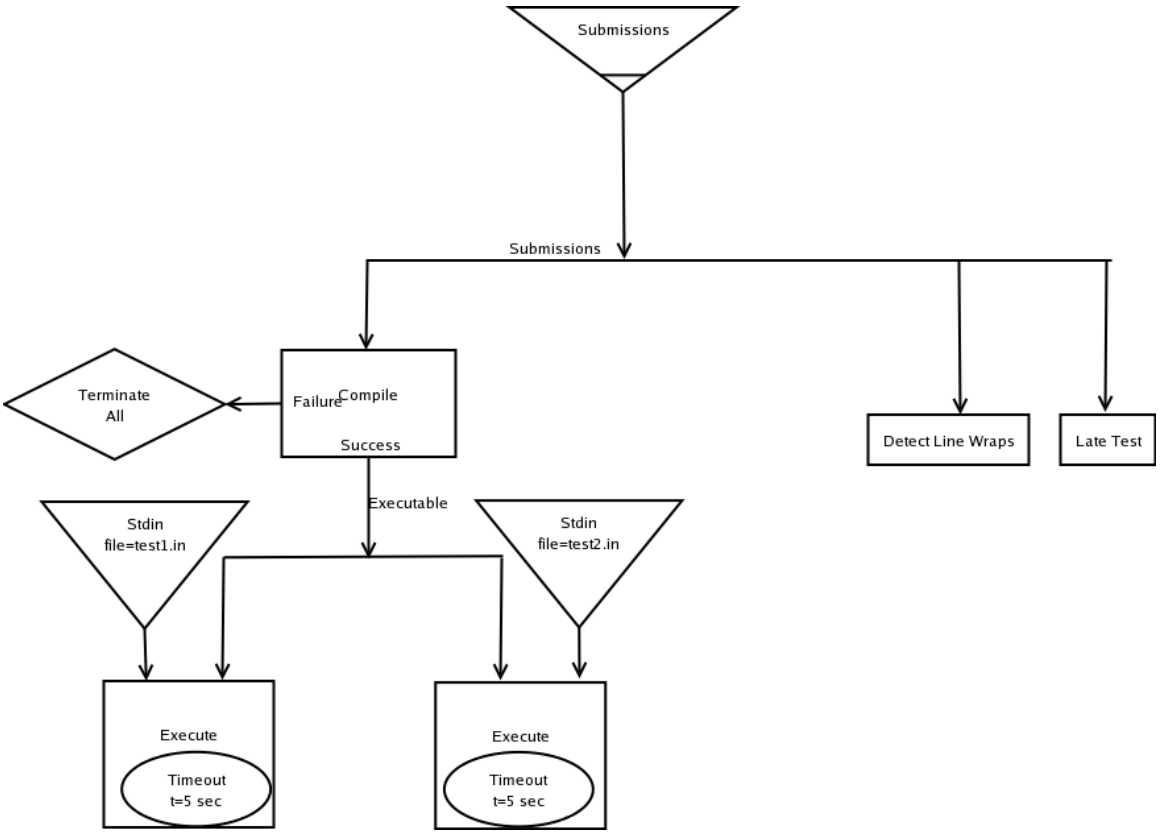


Figure 7.1: Agar2 Rubric Mock-up

# Chapter 8

# Conclusion

Forty-five years of history in the field of Computer-Aided Assessment has failed to provide us with any generalized and usability-driven designs for a CAA system. Considering the relatively small size of the CSE and CAA communities, a fairly significant amount of work has taken place in the field, and on the basis of that work we can easily extract a set of guidelines that are certainly necessary, and hopefully sufficient, conditions for a CAA tool to achieve widespread acceptance. A remarkable number of difficulties both technical and psychological face the would-be implementer of such a system.

In its current incarnation, Agar has proven invaluable for many forms of grading for the users that have overcome its difficulties in interface design. With respect to our initial set of design guidelines (speed, accuracy, quality of feedback, consistency, flexibility), we have met all to some extent. For trained users, all of these requirements are met, although the act of grading using Agar alters the fundamental paradigms used by a grader. For the untrained user, speed and flexibility suffer due to the uncalibrated user models and too much being done invisibly by the system. It is these areas that we hope to improve upon most in Agar2, by allowing a more intuitive interface for the creation of a rubric and a more informative view of the steps being automatically taken on the graders' behalf.

The lessons learned from the historical review, development of Agar, and three academic quarters of support, feature requests, and user feedback on Agar have granted us an invaluable understanding of the problem domain. While usable in its current version and a valuable development in its own right, a re-imagined version of Agar could be the long awaited low-learning-curve, minimally invasive, generalized framework for computer aided assessment that has been lacking since the late 1950s.

# Bibliography


[1] Extreme programming: A gentle introduction. http://www.extremeprogramming.org.

[2] Funderstanding - constructivism. http://www.funderstanding.com/constructivism.cfm.

[3] Mozilla plugindoc. http://plugindoc.mozdev.org/.

[4] wxpython home page. http://www.wxpython.org/.

[5] wxwidgets home page. http://www.wxwidgets.com.

[6] *Lint, a C program checker*, volume 2A, chapter 15, pages 292–303. Bell Laboratories, 1978.

[7] Frank B. Baker. *Fundamentals of Item Response Theory*. ERIC Clearinghouse on Assessment and Evaluation, 2001.

[8] Blackboard learning system. http://www.blackboard.com/products/academic/ls/index.htm.

[9] Fred Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[10] Jerome Bruner. *Actual Minds, Possible Worlds*. Harvard University Press, 1987.

[11] Computer aided assessment: Basic tools for warwick. http://www.warwick.ac.uk/ETS/Publications/Guides/Assessment/basicassess.htm.

[12] James Dalziel. Enhancing web-based learning with computer assisted assessment: Pedagogical and technical considerations. In *Proceedings of International Computer Assisted Assessment Conference*, 2001.

[13] George E. Forsythe and Niklaus Wirth. Automatic grading programs. *Commun. ACM*, 8(5):275–278, 1965.

[14] Free Software Foundation. Bourne-again shell. http://www.gnu.org/software/bash/.

[15] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5):272–275, 1969.

[16] Colin Higgins, Pavlos Symeonidis, and Athanasios Tsintsifas. The marking system for coursemaster. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 46–50. ACM Press, 2002.

[17] Jack Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, 1960.

[18] Lauri Malmi, Ari Korhonen, and Riku Saikkonen. Experiences in automatic assessment on mass courses and issues for designing virtual courses. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 55–59. ACM Press, 2002.

[19] Abelardo Pardo. A multi-agent platform for automatic assignment management. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 60–64. ACM Press, 2002.

[20] Guido Van Rossum. Python programming language. http://www.python.org.

[21] Bruce ”Tog” Tognazzini. *Tog on Interface*. Addison-Wesley, 1996.

[22] Turing’s craft. http://www.turingscraft.com.

[23] Kevin Turner. Writing a gimp plugin. http://gimp-plug-ins.sourceforge.net/doc/Writing/html/plug-in.html.

[24] urs von Matt. Kassandra: the automatic grading system. *SIGCUE Outlook*, 22(1):26–40, 1994.

[25] Larry Wall. Perl programming language. http://www.perl.com.

# Appendix A

# Existing Tools

- *Bulk Tests* - A tool that compares the output of an executable with a provided sample output file. Rather than directly compare the two, it splits on lines of "−"s and treats each as a separate test. Uses the advanced interface to output a number of points equal to the percentage passed times the "maxpoints" argument.
- *Copy File* - A tool that always returns success. It is used to ensure that a given file exists in each students submission directory.
- *C++ Driver Tests* - Replaces the "main" function in a submission with the contents of a given file. Compiles and runs the result. Otherwise behaves the same as Diff Test.
- *Diff Test* - Run an executable with given input and check to see if the output matches a given output file. Options include a timeout and optional flags to the diff utility to allow more imprecise matches.
- *Detect Windows Code* - Check the given file for DOS/Windows style newlines.
- *Edit Distance Test* - Similar to the Diff Test, but utilizes Python's difflib library to check the edit distance between the output of the executable and the sample output.

- *Exit Code* - Returns 0 if the given executable returns 0, 1 otherwise.
- *Failure* - Always return failure. Useful for setting up a single top-level rubric item that contains many sub-tests. Most commonly this is used for a “Penalties” top-level, which contains a Late Test, Detect Windows Code, Detect Line Wraps, etc.
- *Correct Filename* - Check that the filename provided matches the file invoked on. Earlier attempts at CAA had less-robust submission identification systems and so students were requested to submit their code with a given filename. This is less important now This test is also one of the few that should generally use the “Penalize after all fail” option.
- *Post Hand Grading* - Assumes the file it is invoked on is a list of point scores, one line per problem. Uses the advanced interface to output the number of points specified on a given line.
- *Invoke Make* - Invoke “make” in the submission directory and return the results of that operation.
- *Late Test* - Check the full path of the given file for a given token, or LATE if none is specified.
- *Early/Late Points* - Grant a specific number of points for submissions turned in 1, 2, or 3 days before a given deadline, and 1, 2, or 3 days after. Uses the advanced interface.
- *Detect Line Wraps* - Detects lines of code that are greater than 80 characters long, when tabs are expanded. The size of the tab stops and the max line length may be specified
- *Success* - Returns success. Used primarily for things that are graded by hand to give students all points by default and then reduce the score with comments.

- *Post OCR* - Before the Marksense system was split out from Agar, this was used to grade a file of student response against a given answer key, one answer per invocation. Not generally used now.
- *Regexp Code* - Check the given file for matches of the given regular expression.

# Appendix B

# Version History

The CVS logs for the project over the project's lifespan really give direct evidence and insight into the development cycle. About one in three CVS updates was immediately propagated to the deployed version of Agar. These logs have been augmented with ChangeLog information from within GraderFrame.py whenever the ChangeLog is more informative.

```
----------------------------
revision 1.35
date: 2004/04/30 20:58:19;  author: titus;  state: Exp;  lines: +151 -8
* Added an option to pass flags to diff in the Diff Test tool.
* Got relative path storage working, so workspaces can now be shared in many
  cases.
* Save dialogs give better default filenames
* Fixed bug where LATE files are not treated as final submission files
* Fixed bug where filenames are displayed incorrectly in Submission tree
* Added a listing of the Tool's original name to the Properties Dialog (buis)
```

```
* Added a button to the Properties Dialog that allows the options to
  be re-read from the tool executable (eharris)
----------------------------
revision 1.34
date: 2004/04/14 20:13:06;  author: buis;  state: Exp;  lines: +6 -9
*** empty log message ***
----------------------------
revision 1.33
date: 2004/04/12 21:05:19;  author: titus;  state: Exp;  lines: +150 -35
* Altered the Drag-and-Drop system for the Rubric so tests can be
  re-ordered.
* Added a number of options to the context menu for Rubric tests.
* Fixed where the context menu appears.


----------------------------
revision 1.32
date: 2004/04/07 18:46:27;  author: titus;  state: Exp;  lines: +53 -17
A few bug fixes, I'm posting an update to the server, so this is 0.3.7.
* Fixed segfault when hierarchical tests have been removed before running.
* Added tool for copying known files into submission directories
* Sorted toolbox
* Fixed behavior of default filemask
----------------------------
revision 1.31
```

```
date: 2004/03/09 17:28:49;  author: titus;  state: Exp;  lines: +15 -2
Undid work on using emacserver
----------------------------
revision 1.30
date: 2004/03/09 17:26:05;  author: titus;  state: Exp;  lines: +48 -2
*** empty log message ***
----------------------------
revision 1.29
date: 2004/03/06 16:06:58;  author: titus;  state: Exp;  lines: +0 -1
Removed erroneous testing garbage.
----------------------------
revision 1.28
date: 2004/02/29 21:07:51;  author: titus;  state: Exp;  lines: +206 -38
Lots of changes, mostly little things regarding UI and the Submission
Manager.  A couple additional Context-Menu options for Submissions.
* Added confirmation request before sending mailbacks.
* Hardwired the "Success" and "Failure" tests to be faster and work
  even in the face of weird filemasks.  I don't really like this but it
  seems to be mostly what people want. I reserve the right for this
  to change in the future.
* Removed the "N/A" result for tests that were never invoked on a submission
  for lack of files matching the filemask. Now the test simply fails
  (this makes it simpler to deal with comments overriding this behavior.)
* The OCR Wizard and the Written Work Wizard now automatically invoke
```

"Run Tests"

* We've got viewers configured for .doc (oowriter and abiword)
* Submission Manager now appears in the Actions menu.
* By default we will attempt to open only a single emacs window (using the emacsserver extensions.)
* The context-menu for Submissions on the Grading tab now has a "Rerun Tests on Sub" option to rerun the tests for ONLY this submission.
* If submissions cannot be understood automatically, the option to invoke the Submission Manager is on the warning dialog.

---

revision 1.27

date: 2004/02/25 04:44:06;  author: titus;  state: Exp;  lines: +6 -0

Nothing important, doing a bugfix on usernames not exporting correctly because of /'s

---

revision 1.26

date: 2004/02/17 06:36:30;  author: titus;  state: Exp;  lines: +23 -9

Minor bugfixes, some new functionality in difftest / editdist, added new pointlate tool to deal with early/late submission points slightly more robustly.

---

revision 1.25

date: 2004/02/12 22:07:02;  author: titus;  state: Exp;  lines: +0 -1

```
Fixed vestigal code error.

----------------------------

revision 1.24

date: 2004/02/12 21:46:28;  author: titus;  state: Exp;  lines: +2 -0

Merging diffs, no real changes.

----------------------------

revision 1.23

date: 2004/02/11 05:07:48;  author: titus;  state: Exp;  lines: +123 -12

Many little bug fixes, some mild tweaking to the saving / loading code,

and a fixing of my improper use of TreeCtrl's and iterating through such.

----------------------------

revision 1.22

date: 2004/02/06 03:55:21;  author: titus;  state: Exp;  lines: +1 -1

Fixed drag-and-drop comment bug, the useMake compile bug.

----------------------------

revision 1.21

date: 2004/02/06 03:50:27;  author: titus;  state: Exp;  lines: +23 -9

Updating, fixed a comment bug and one of the terminal bugs.

----------------------------

revision 1.20

date: 2004/02/03 21:20:25;  author: leviee;  state: Exp;  lines: +10 -1

Added error message in case exporting sheet fails

* Added dialog boxes confirming most previously silent operations (like

  writing mailbacks.)
```

```
* Bug-fixes
----------------------------
revision 1.19
date: 2004/01/25 20:57:19;  author: titus;  state: Exp;  lines: +4 -1
Fixed startup wizard bug.
----------------------------
revision 1.18
date: 2004/01/25 20:52:08;  author: titus;  state: Exp;  lines: +15 -13
Minor bugfix.
----------------------------
revision 1.17
date: 2004/01/25 20:40:47;  author: titus;  state: Exp;  lines: +861 -299
Biggest . . . update . . . ever.

See GraderFrame.py for changelog, it's seriously huge.  Stability up,
feature set up.  I'm so happy.
* Added checkbox to New Comment dialog to auto-assign that comment to the
  current submission.
* Changed the submission list on the Grading tab to be a Tree, each submission
  contains the files of that submission as leaves.
* Right clicking the Comment Tree or the Submission Tree on the Grading tab
  brings up a context menu.
* The context menu on the Comment tree allows comments to be sorted in ways
  other than just alphabetically (sort by recently used, frequency, or rubric
```

item).

* Editors other than just Emacs are possible now.
* The context menu on submissions has a ’Run Executable’ option, which runs the executable for the submission in an xterm and waits upon program termination for an extra newline so you have time to look over the results.
* The Compile Dialog now has an option to attempt using Make when possible, and then fall back to compiling the ’old fashioned’ way if no makefile is present or if the make build fails.
* Assigned comments now show their point values.
* THE SEGFAULT ON RUN TESTS BUG IS GONE!
* Saving a ’template’ (that is, a rubric or rubric & comments with no submissions) now works properly.
* Fixed a bug with loading from the command line.
* Added the status bar at the bottom, mostly to notify you when a Save has been successful.
* Agar now autosaves.  If you’ve already specified a filename, it will save in filename + ’~’, otherwise it saves in autosave.agw in the directory you invoked Agar in.  The autosave happens every minute.
* Point values not longer have to be integers, they can now have one significant figure after the decimal.
* No longer requires Ctrl+C to terminate when started using the Startup Wizard

---

revision 1.16

```
date: 2004/01/22 20:32:05;  author: titus;  state: Exp;  lines: +47 -74
Fixing my CVS errors still.
----------------------------
revision 1.15
date: 2004/01/22 20:24:00;  author: titus;  state: Exp;  lines: +7 -4
I think I fixed the evil segfault, and I'm pretty sure I've gotten
usernames to look like usernames (no trailing '/'s) for all of the
3 major modes.
----------------------------
revision 1.14
date: 2004/01/20 05:33:43;  author: titus;  state: Exp;  lines: +30 -3
Fixed a bunch of little things, added Bulk Tests and Invoke Make.  Most
important was the squashing of Comment Dialog bugs, and a couple minor
extensions of the Submission Manager.
* Added some extra functionality to SubmissionManager.
* StartupWizard now tests to see if you have write access to the directory
  you identify as the base turnin directory.  If not, it asks if you would
  like to copy the submissions over to somewhere else.
* Added an 'Invoke Make' tool and a 'Bulk Test' tool.  Bulk test was a lazy
  out on my part when doing a fast-pass at grading today's CS 14 assignment.
  It lets you test sections of the output of an executable against a sample
  output, and collect points equal to the number of those test sections that
  match.
* Fixed bugs in Comment Dialog that prevented proper editing.
```

```
* Added a 'Tool Help' button on Configuration Dialogs for rubric tests.
  The help button displays the output from the tool when given the --help
  parameter.


----------------------------
revision 1.13
date: 2004/01/19 22:33:46;  author: titus;  state: Exp;  lines: +2 -0
Got rid of the "can't edit a comment" bug.
----------------------------
revision 1.12
date: 2004/01/05 06:43:10;  author: titus;  state: Exp;  lines: +33 -18
A much much better Submission Manager.  Submission Finder is now
deprecated.
* Removed SubmissionFinder, replaced with Submission Manager, which
  allows much greater control over the files in a given submission,
  as well as parsing of Makefiles to find relevant files to include
  in a submission.
----------------------------
revision 1.11
date: 2003/12/11 20:51:10;  author: titus;  state: Exp;  lines: +2 -2
Oh yeah, and fixed the XPDF popping up along with emacs problem.
----------------------------
revision 1.10
date: 2003/12/11 20:41:08;  author: titus;  state: Exp;  lines: +21 -2
```

```
Fixed some of the filemask config annoyances and assignment name bugs that
were bothering me
----------------------------
revision 1.9
date: 2003/12/11 19:49:11;  author: titus;  state: Exp;  lines: +2 -4
Nothing special.
----------------------------
revision 1.8
date: 2003/11/25 07:16:02;  author: titus;  state: Exp;  lines: +1 -1
Minor tweaks.
----------------------------
revision 1.7
date: 2003/11/21 00:22:54;  author: titus;  state: Exp;  lines: +2 -1
Fixed configuration dialog.
----------------------------
revision 1.6
date: 2003/11/19 03:10:07;  author: titus;  state: Exp;  lines: +1 -1
Fixed executable and where submissions come from.
----------------------------
revision 1.5
date: 2003/11/18 06:53:48;  author: titus;  state: Exp;  lines: +1 -1
Fixed bug from bad merge.
----------------------------
revision 1.4
```

```
date: 2003/11/18 06:45:40;  author: titus;  state: Exp;  lines: +120 -84
Lots of updates.
* Added StartupWizard.  Run agar.py from the command line
  without a directory argument.  Useful for written work, processing OCR
  results, or grading things other than C/C++.
* Added configuration storage.  The file ~/.agar.cfg stores record of
  your most recent settings.  This is somewhat experimental, if you suffer
  from inexplicable behavior, try removing this file.  If that solves your
  problem, PLEASE send an email describing the issue to titus@cs.ucr.edu.
* Fixed Drag-And-Drop.  All Drag-And-Drop is now done with the left button,
  rather than the right button.
* Added versioning and update messages.
* Changed executable name from grader.py to agar.py.
* Added Point configuration to the WrittenWork Wizard, and fixed the comment
  values for the same.
* Revised the saving/loading system, no more AGT's.  Now all save files
  describe what information they contain.  The first time you save a workspace,
  or whenever you hit Save As, you will be given a dialog that asks what
  information you want to save (Rubric, Submissions, Comments, Misc).
----------------------------
revision 1.3
date: 2003/11/18 00:31:59;  author: titus;  state: Exp;  lines: +7 -6
Startup wizard works, WrittenWizard does a point configuration that is
sweeeet, OCR Wizard works properly, the advanced point system has been
```

```
tweaked to still deal with statistics as it ought to.

----------------------------

revision 1.2

date: 2003/11/17 07:40:31;  author: titus;  state: Exp;  lines: +56 -11

I think the new version is ready-to-go-ish

----------------------------

revision 1.1

date: 2003/11/12 19:59:45;  author: titus;  state: Exp;

branches:  1.1.1;

Initial revision

----------------------------

revision 1.1.1.1

date: 2003/11/12 19:59:45;  author: titus;  state: Exp;  lines: +0 -0

A new era begins, major revisions, and a new CVS repository.


=============================================================================


----------------------------

revision 1.45

date: 2003/11/06 20:58:22;  author: titus;  state: Exp;  lines: +71 -4

Fixed some comment issues and the issue with the Wizards allowing

duplicated names for rubric items.

----------------------------

revision 1.44
```

```
date: 2003/11/06 05:04:28;  author: titus;  state: Exp;  lines: +28 -0
Primitive support for "advanced" tool interface.
----------------------------
revision 1.43
date: 2003/11/05 21:03:35;  author: titus;  state: Exp;  lines: +143 -1
Added WrittenWork wizard
----------------------------
revision 1.42
date: 2003/11/05 18:39:06;  author: titus;  state: Exp;  lines: +46 -18
A couple tweaks here and there.  Need to add support to the PostOCR wiz
for doing Tom-style evil MC grading (1 for good, 0 for blank, -1 for wrong.)
----------------------------
revision 1.41
date: 2003/11/03 05:10:59;  author: titus;  state: Exp;  lines: +181 -23
Added Post-OCR wizard support.
----------------------------
revision 1.40
date: 2003/10/21 17:40:39;  author: titus;  state: Exp;  lines: +9 -20
Mail works now.
----------------------------
revision 1.39
date: 2003/10/16 16:57:17;  author: titus;  state: Exp;  lines: +7 -1
Finishing up my cleanup of the GUI, back to functionality in a bit.
----------------------------
```

```
revision 1.38
date: 2003/10/16 05:17:38;  author: titus;  state: Exp;  lines: +58 -9
Fixed a few little UI things that were bugging me.
----------------------------
revision 1.37
date: 2003/10/14 07:47:54;  author: titus;  state: Exp;  lines: +0 -4
Aesthetic deletions.
----------------------------
revision 1.36
date: 2003/10/14 03:08:25;  author: titus;  state: Exp;  lines: +5 -1
Regexp now works.
----------------------------
revision 1.35
date: 2003/10/10 23:15:28;  author: titus;  state: Exp;  lines: +10 -3
Export does so by row correctly, in the correct column order, grader
supports pdf limitedly.
----------------------------
revision 1.34
date: 2003/10/09 04:18:29;  author: titus;  state: Exp;  lines: +3 -2
Changed the progress dialog to be a bit more interesting.
----------------------------
revision 1.33
date: 2003/10/06 07:23:02;  author: titus;  state: Exp;  lines: +200 -42
Now add a comment whenever a test passes / fails.
```

```
----------------------------
revision 1.32
date: 2003/09/30 22:45:36;  author: titus;  state: Exp;  lines: +4 -4
Slight changes, need to test export.
----------------------------
revision 1.31
date: 2003/09/30 20:45:22;  author: titus;  state: Exp;  lines: +12 -5
Modified export format.
----------------------------
revision 1.30
date: 2003/09/21 05:06:15;  author: titus;  state: Exp;  lines: +9 -6
Mildly documented for release.
----------------------------
revision 1.29
date: 2003/09/18 16:48:12;  author: titus;  state: Exp;  lines: +1 -1
Works with the hanoi stuff the new grads turned in yesterday.
----------------------------
revision 1.28
date: 2003/09/18 05:31:29;  author: titus;  state: Exp;  lines: +68 -35
Beta release version.
----------------------------
revision 1.27
date: 2003/09/18 02:07:25;  author: titus;  state: Exp;  lines: +1 -1
Mild changes.
```

```
----------------------------
revision 1.26
date: 2003/09/17 17:59:15;  author: titus;  state: Exp;  lines: +220 -71
Regular checkin.
----------------------------
revision 1.25
date: 2003/09/10 23:05:13;  author: titus;  state: Exp;  lines: +237 -38
*** empty log message ***
----------------------------
revision 1.24
date: 2003/09/10 05:13:57;  author: titus;  state: Exp;  lines: +402 -216
Still needs a configuration dialog box, menu shortcuts, tooltips on a few
things, and better filename handline.  But we've got 95\%.
----------------------------
revision 1.23
date: 2003/09/07 05:27:05;  author: titus;  state: Exp;  lines: +291 -509
Mostly "done."  Mailing back mailbacks doesn't work yet (not sure why),
there are a couple minor lookup bugs going on, but beyond that it's just
feature addition.
----------------------------
revision 1.22
date: 2003/08/30 07:15:03;  author: titus;  state: Exp;  lines: +298 -123
Whole helluva lot of stuff done.  All the base functionality is here
I think, its just bug fixes and feature adding from here on in.
```

```
----------------------------
revision 1.21
date: 2003/08/29 23:59:01;  author: titus;  state: Exp;  lines: +20 -19
Minor updates
----------------------------
revision 1.20
date: 2003/08/29 21:49:44;  author: titus;  state: Exp;  lines: +306 -193
I think I just broke through into the final phase.
----------------------------
revision 1.19
date: 2003/08/29 06:30:40;  author: vinh;  state: Exp;  lines: +156 -153
Added functionality to add new comments.  Added classes Comment and
CommentDialog.
----------------------------
revision 1.18
date: 2003/08/28 19:51:42;  author: titus;  state: Exp;  lines: +151 -17
Manual override tweak for grading.
----------------------------
revision 1.17
date: 2003/08/28 02:47:13;  author: titus;  state: Exp;  lines: +11 -12
Got configuration options for tools to be persistant.
----------------------------
revision 1.16
date: 2003/08/26 07:22:44;  author: vinh;  state: Exp;  lines: +27 -0
```

Added sync’ed selection on Scratchpad tab.  Sync’ed scrollbar doesn’t work yet.

---

revision 1.15

date: 2003/08/25 23:51:43;  author: titus;  state: Exp;  lines: +150 -104

Done for the day.  Points are now displayed and calculated.

---

revision 1.14

date: 2003/08/25 05:42:41;  author: titus;  state: Exp;  lines: +64 -20

Going to bed.  Updated interface a bit, added combobox for point-types,

added requisit parts to Tool class for dealing with that and began work

on tracking when a tool/test has changed and thus could be re-run efficiently.

Need to break the functionality from onRun and a couple of the other long

functions into small pieces so they can be more modular / easily recalled.

---

revision 1.13

date: 2003/08/25 01:18:50;  author: titus;  state: Exp;  lines: +117 -102

Fixed a couple bugs, made the settings dialogs for the tools a bit cleaner,

minor tweaks here and there.  Fixed a number of TODOs.

---

revision 1.12

date: 2003/08/24 22:32:38;  author: titus;  state: Exp;  lines: +148 -49

Actually able to do some minimal grading, except for not calculating

scores or anything like that yet.

---

```
revision 1.11
date: 2003/08/23 20:56:42;  author: titus;  state: Exp;  lines: +112 -9
Beginning to rough in the Compiler tool.
----------------------------
revision 1.10
date: 2003/08/23 19:15:27;  author: titus;  state: Exp;  lines: +180 -29
Tabs 1 and 3 are mostly done, I need to get a special tool for compiling
and develop a system for having tools run on the executable, but the
base functionality is very close.
----------------------------
revision 1.9
date: 2003/08/23 05:46:51;  author: titus;  state: Exp;  lines: +172 -69
Commented and with minor interface tweaks.
----------------------------
revision 1.8
date: 2003/08/23 02:52:49;  author: titus;  state: Exp;  lines: +125 -122
Beginnings of tracking points per tool test, also now displaying results
from tool tests.
----------------------------
revision 1.7
date: 2003/08/22 22:28:30;  author: titus;  state: Exp;  lines: +249 -16
I can now autograde (sort-of) whether an set of submissions has line wraps.
----------------------------
revision 1.6
```

```
date: 2003/08/21 01:36:06;  author: titus;  state: Exp;  lines: +11 -8
Mostly done . . . still.
----------------------------
revision 1.5
date: 2003/08/21 00:24:23;  author: titus;  state: Exp;  lines: +64 -22
Tools being read in from tools directory, properties nearly working,
need to work on aliasing still.
----------------------------
revision 1.4
date: 2003/08/15 20:01:37;  author: titus;  state: Exp;  lines: +141 -34
Lots of work done on the rubric section.
----------------------------
revision 1.3
date: 2003/08/13 01:34:00;  author: titus;  state: Exp;  lines: +42 -1
Got some drag-and-drop working for the Rubric.
----------------------------
revision 1.2
date: 2003/08/12 17:14:46;  author: titus;  state: Exp;  lines: +9 -0
Added todo list.
----------------------------
revision 1.1
date: 2003/08/12 00:08:14;  author: titus;  state: Exp;
First commit with wxPython attempt.
=============================================================================
```