Computational Quiz Generation

Daniel Hoffman, Felix Giannelia and Ming Lu
Department of Computer Science
University of Victoria
PO Box 3055 STN CSC
Victoria, BC V8W 3P6
Telephone: 250-472-5768
FAX: 250-472-5708
Email: {dhoffman,niteling,luming}@cs.uvic.ca

***Abstract*—Every computer science instructor would like to have quizzes which can be delivered online, with the answers captured and marked automatically. Online quizzes are useful in traditional lecture courses, more useful in online courses and extremely useful in massively online open courses (MOOC). While multiple choice quizzes are already widely used, they are limited. In particular, they are often not effective in evaluating procedural mastery, e.g., tracing a C program. We present *CQG*, a novel online quiz generation framework focusing on evaluation of procedural mastery. The framework includes code and detailed processes for the development of new questions and new question types. The question type development process is based on a reference architecture and a verification procedure aimed at minimizing server crashes and marking errors. *CQG* has been used in first, second and third-year courses. Twenty-three quizzes have been developed, focusing on code tracing in C, C++, Java and Python, on procedures commonly taught in computer networking courses—such as computing a Hamming code—and on the classic encryption and decryption algorithms.**

Keywords-code reading; active learning; web-based tool

I. Introduction

Online quizzes relieve the instructor of the tedium of marking exercises: every instructor's dream. In courses with a traditional lecture format, online quizzes are helpful, especially if the class is large. In courses with online delivery, online quizzes are a perfect fit. We believe that the biggest impediment to the widespread use of massively open online courses (MOOC) is the development of much better online evaluation.

Online quizzes are in use today, primarily in two formats. Multiple choice quizzes are easy to create and deliver and, while limited, are effective for some skill evaluations. At the other extreme are sophisticated online simulations or labs. This approach provides impressive capabilities but is expensive to develop, deliver and maintain.

We present Computational Quiz Generation (*CQG*), a web application framework exploiting the opportunity between multiple choice and online labs/simulations. With this approach, there are four fundamental steps:

1) The instructor selects a computation.
2) The instructor creates a question, in template form, by replacing one or more items with placeholders. We call the placeholders "hotspots" and say that this step "activates the computation."
3) The student fills in the hotspots, producing a completed template.
4) The application automatically checks the student solution by performing the computation on the completed template.

We illustrate these steps with a well-known procedure: adding a column of integers. Figure 1(a) contains a completed addition problem. In Figure 1(b) the two digits in the sum have been replaced by hotspots a and b. The student provides values for a and b; obviously, the solution is correct if and only if $a = 7$ and $b = 0$

This simple example illustrates several important points about the *CQG* procedure:

1) So called "backward questions" are possible, where the hotspots are inputs to the computation. Note that, in a backward question, the correct answer may differ from the original. For example, in Figure 1(c), the solution $c = 6$ and $d = 4$ is also correct.
2) While step 2 guarantees that the question is solvable, many selections of digits and hotspots are not solvable, as shown in Figure 1(d).
3) Step 4 requires answer *checking* not answer *generation*; the latter is often much harder.
4) The complete computation and the templates derived from it form an abstraction hierarchy. Figures 1(b) and (c) are more abstract than Figure 1(a). Figure 1(e) is more abstract than Figures 1(a–d).

CQG is a web application implemented with HTML forms technology. Each question is presented as a template. The student provides answers in HTML input elements, such as text boxes and pull-down lists, and clicks a "check" button. *CQG* checks the completed template and issues a message: "correct" or "incorrect".

Section 2 illustrates *CQG* from the student point of view, with detailed discussion of four questions and three question types. Section 3 explains how to use *CQG* to generate many questions from a single original computation and how to produce reusable quizzes from a large pool of questions. Section 4 presents the main server design goals, and discusses

2	3
4	7
7	0

(a) Original

2	3
4	7
a	b

(b) Traditional

2	c
4	d
7	0

(c) Backward

2	0
4	d
7	3

(d) Unsolvable

2	c
4	d
a	b

(e) Abstract

Fig. 1. Question generation steps: adding a column of integers

both the server failures encountered to date and the methods developed to eliminate those failures. Section 5 summarizes our experience with *CQG*. Section 6 presents the related work in quiz generation in computer science.

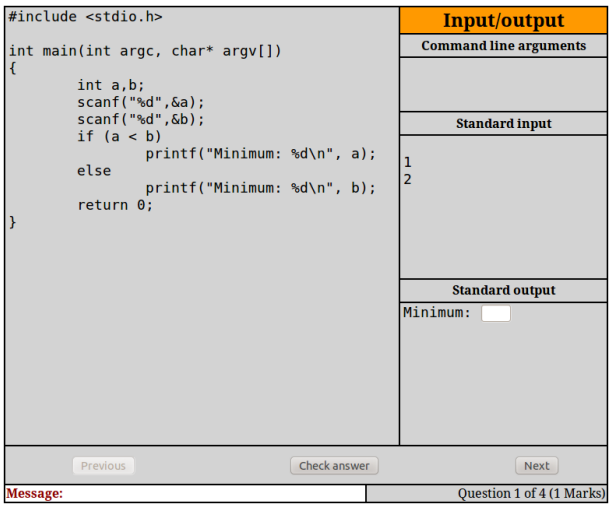
II. Student View

In the *CQG* system, a quiz consists of a list of questions. Each question has a question type; fourteen question types are currently provided.

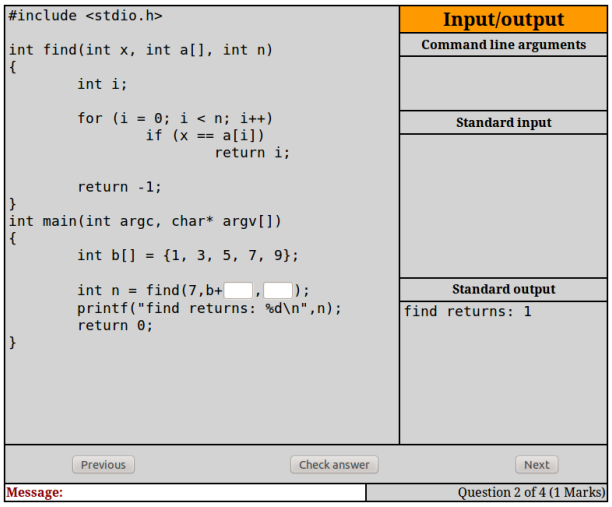
A. Active code reading

Many instructors believe that students would write better code if they spent more time reading code; many instructors already do ask students to read code. Unfortunately, the reading tends to be unfocused and little is learned. We lack precise goals and practical evaluation mechanisms for code understanding. Many students simply do not know how to learn much about a program by reading it.

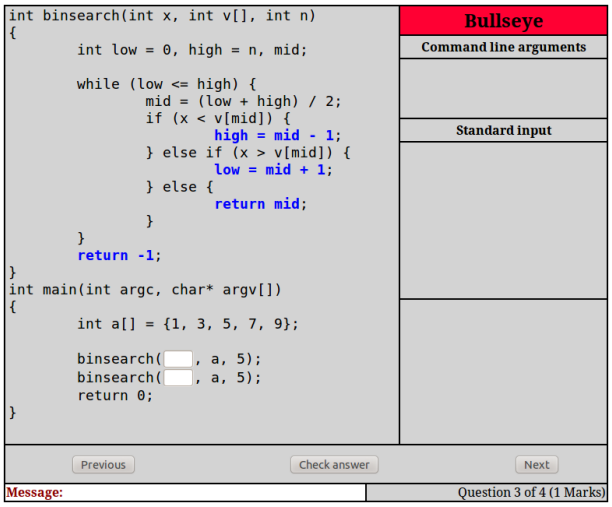
Active code reading is a special case of active learning. Figure 2(a) shows a question of type *Input/output*, designed to support active code reading. On the left side is a complete C program; on the right are screen cells for command-line arguments (not used in this question), standard input and standard output. The student must complete the text field in the standard output cell and press Check answer. The answer is correct if the standard output shown on the screen, including the textbox value, is identical to the standard output from an actual execution.



(a) Minimum of two integers: forward



(b) Linear search: backward



(c) Binary search: code coverage

Fig. 2. The *CQG* application: Code Activator question types

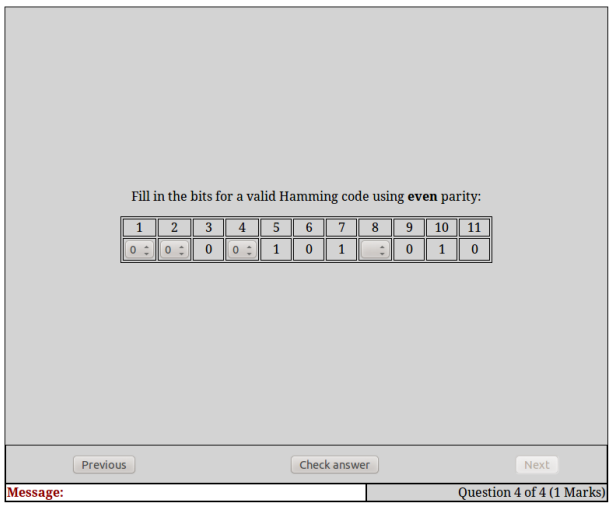


Fig. 3. The *CQG* application: Hamming Code question type

Figure 2(b) shows an *Input/output* question focusing on linear search using "slices" of C arrays. In the call to `find`, the student may alter only the displacement d and length n of the slice `b[`$d..d+n-1$`]`. The student is asked to complete an invocation of `find` such that the value 7 is located in position 1 of the slice. In a correct answer $d = 2$ and n is 2 or 3.

Figure 2(c) illustrates the *Bullseye* question type. Again, active code reading is the goal. There are two changes in screen layout from *Input/output*: (1) there is no standard output cell and (2) there are one or more highlighted lines in the code. The student must complete the text fields such that each highlighted line is executed at least once. In Figure 2(c), there are four highlighted lines. The function `main` invokes `binsearch` twice, with a textbox for the target element `x`. The net effect is that the student must choose two `x` values which achieve 100% statement coverage of `binsearch`.

CQG supports a third active code reading question type. With *Find-the-failure*, the code cell contains a specification of a function F, a purposely faulty implementation of F, and an invocation of F. The student must supply failure-revealing values, i.e., values which cause the standard output to be incorrect with respect to the specification.

While C is used in the examples in this section, C++, Java, and Python are also supported.

B. A Hamming code question type

CQG question types need not be based on programming language code, as shown by the *Hamming* question type in Figure 3. In a Hamming codeword, the bits in positions $1, 2, 4, 8, \ldots$ are check bits; the other bits are message bits. The check bits are computed using a generalized parity scheme. In Figure 3, the student has used the pull-down lists to enter the first three of the four check bits. The answer is correct if the completed form is a correct Hamming code.

C. Quiz modes

The tool is designed for two types of quiz: practice and marked.

1) In a practice quiz, the student is informed immediately whether his/her answer is correct. No record is kept of the student's work.
2) In a marked quiz, the student must log in before beginning the quiz. While no immediate feedback is given regarding the correctness of his/her answers, the answers to each question are logged. Marks are computed automatically from the log.

III. Question author view

Our goal is to develop quizzes which are usable by students side-by-side in a lab and reusable by students term by term. Such quizzes require a large pool of questions. Creating good automated quiz questions is hard, requiring skill and experience, as well as refinement through repeated use and improvement. Many instructors have the ability and desire to develop such questions but do not have enough time unless the questions can be generated.

A. Question generation from templates

Question generation in *CQG* is based on templates. For example, the question shown in Figure 2(a) was generated from the template shown in Figure 4. Generation is implemented in Python; in actual use, the question author assigns the values shown in Figure 4 to Python variables. The C code is separated into global code and main code to make it easier to generate the extra code occasionally needed to handle answer checking. Following the code templates are templates for command-line arguments, standard input and standard output. Each of the five templates can contain a mixture of plain text and markers for hot spots: locations which might contain a text field in a delivered question. In Figure 4, the markers `$x0`, `$x1`, and `$y0` are used; a marker can be any string which does not already occur in the templates.

The list of tuples in Figure 4 specifies how the hot spots are to be filled in: with a string or numeric constant, or with `None`. The latter indicates that the hot spot should appear as a text field. Since there are five entries in the tuple list, this template will generate five questions. The question in Figure 2(a) was generated from the tuple `[1,2,None]`.

The third tuple, `[None,None,1]`, generates a backward question. The standard input contains a textbox for each of `a` and `b` and the standard output (`1`) is provided; the student must supply `a` and `b` values such that the output is `1`. Forward questions test procedural mastery in a straightforward way. With patience and knowledge of C, any forward question can be answered correctly. Backward questions often require more sophisticated thinking. Viewed another way, while *CQG* could generate the answer for any forward *Input/output* question, generating answers for backward questions is an undecidable problem. Because *CQG* checks only completed templates, the undecidability of answer generation is not an issue.

Global code

```
#include <stdio.h>
```

Main code

```
int a,b;
scanf("%d",&a);
scanf("%d",&b);
if (a < b)
   printf("Minimum: %d\n", a);
else
   printf("Minimum: %d\n", a);
```

Command-line arguments

Standard input

```
$x0
$x1
```

Standard output

```
Minimum: $y0
```

Tuple list

```
[1,2,None]
[9,3,None]
[None,None,1]
[None,None,20]
[None,None,-2]
```

Fig. 4. Question template for *Input-output*

Figure 4 illustrates one use of the template abstraction hierarchy. Each *CQG* question is a template to be completed by the student. Each question template, such as the one shown in Figure 4, is a more general template, completed partially and repeatedly by the author.

The question in Figure 2(b) was generated from a template with four hotspots: the three parameters to `find` plus the return value in the standard output: `[`x`,`d`,`n`,`y`]`. From that template, the tuple `[7,None,None,1]` generated the question in Figure 2(b). Dozens of four-tuples can be generated with cut-and-paste in a few minutes. For example, in increasing order of difficulty are forward questions (1) with $d = 0$ and $n = 0$, (2) with one of d and n non-zero, and (3) with both d and n non-zero.

The question in Figure 2(c) was generated from a template with six hotspots: one for each of the four target lines and the target elements in each of the two `binsearch` invocations. From that template, the tuple `[True,True,True,True,None,None]` generated the question in Figure 2(c). Easy questions contain only a single targeted line while harder questions include more targeted lines.

The question in Figure 3 was generated from a significantly different kind of template. Here, each template contains one message bitstring, and a list of hotspot indexes for each generated question. The indexes are (a) one-relative and (b) refer to the code word generated from the message bitstring, not the message itself. The forward question in Figure 2(d) was generated from the message `'0101010'` and index list `[1,2,4,8]`. Easy questions are forward and contain very few hotspots. While it is easy to generate a few Hamming Code questions by hand, the template format makes tuple generation attractive. Because the template file is written in Python, arbitrary Python code can be embedded. Consider the Python function `g(`m`,`n`)` which takes message string m and generates all index lists of length n into the code word. This function is easy to implement in Python, and can be used to generate large numbers of Hamming questions.

B. Quiz specification with random selection

Support is provided for packaging questions in quizzes. The author can simply provide a list of questions, along with a mark for each; each student will see the same quiz. Given a sufficient collection of templates, however, it is easy to create a "cheat-proof" quiz, where copying from a neighbor is no help at all. To accomplish this, the author specifies a list of question groups. Each group consists of a set of pedagogically equivalent questions and a number N, indicating the number of questions to be randomly sampled from the group. Each time the quiz is begun by a student, the sampling is done again.

Consider a quiz constructed from the five questions generated from the template in Figure 4. Suppose that the author chooses two groups: the two "forward" questions and the three "backward" questions. A two-question quiz could be constructed by selecting one question from each of the two groups. Each time a student begins the quiz, he or she will receive one of the six possible quizzes.

CQG automatically collects data on student performance including each submitted answer and the time spent on the question.

IV. Design and verification

A. Design goals

Aside from fast question generation and reasonably fast question type development, there are three interdependent *CQG* design goals:

1) *Minimal by design*. We use no clip art, no JPEG images and no animation. Our intention is to focus the student's attention entirely on the activated computation itself.
2) *Low server load*. A new question retrieved from the server typically contains a few hundred bytes. A client answer submitted to the server is typically under 100 bytes. Answer checking is usually very fast. For example, in a C *Input/output* question, the code is precompiled. The *Hamming* answer checking is very fast for the short bitstrings appropriate for quiz questions.
3) *Secure*. *CQG* has been designed to be secure against attack because it is certain that it will be attacked. HTML forms [1] are used with no client-side embedded

code, e.g., no Javascript. HTML pages are generated on the server side using the web2py [2] framework which provides good support for secure servers.

All answers are type checked to avoid failures during answer checking. For example, in an *Input/output* question, textbox entries are restricted to string and numeric constants, making the answer checking immune to code insertion attacks. *Hamming* questions use pulldown lists offering 0 and 1 only. In both question types, the restrictions are checked server side, to protect against hacking of HTML GET/POST parameters.

B. Question type and question template verification

As in all software projects, *CQG* has seen a number of runtime failures. While the software is not safety-critical, the failures are costly. The 24-by-7 availability of online quizzes is lost; failures during marked quizzes are especially problematic.

We focus on failures due to (1) faults in a question type implementation and (2) faults in a question template. As is common in Python code, question type implementations make extensive use of nested lists and dictionaries. Almost all failures were due to list or dictionary dereferencing errors.

We have attacked question type implementation failures with active code reviews. In each source file, each list or dictionary dereference is considered. The reviewer is asked to prove, informally, that the dereference is guaranteed to be legal. Usually the proof cannot be carried out unless certain preconditions are present. In practice, the proofs are usually short and simple. Most of the effort is consumed by writing and maintaining the preconditions.

Most template failures appeared as questions which were either unsafe for one or more answers, or unsolvable. For example, the question shown in Figure 2(b) is unsafe. Because array `b` has 5 elements, if a student enters, e.g., `100` in the first textbox, the program will attempt to reference addresses far outside the memory allocated for `b`. As a result, the behaviour of the code is unpredictable. On the other hand, if the author changes the standard output value to, e.g., `2`, then the question is unsolvable. In simple templates with just a few tuples, the generated questions are usually safe and solvable. In complex questions with a lot of tuples, we often see a few unsafe or unsolvable questions.

To ensure that questions are solvable, we usually use the approach illustrated in Figure 1: each question is derived from a solved computation by adding hotspots. While it is relatively easy to avoid unsolvable questions, unsafe questions are problematic. In particular, for each question type, a process for demonstrating question safety must be developed. In the case of C *Input/output* questions, the template author must prove, typically manually, that any generated question will be safe. Recall that the question in Figure 2(b) is unsafe because entries in the first (slice displacement) or second (slice length) textbox may cause dereferencing errors. The template author can avoid these problems by (1) activating only one of the displacement and length fields in any given question and (2) using a list box rather than a text box to restrict the displacement and length to safe values. Of course, with C++, Java, or Python questions, exception handlers can be used to achieve safety more elegantly.

With the *Hamming* question type, safety is guaranteed. The generator checks that the message is a non-empty bitstring and that the indexes are in range. Answers are provided using listboxes offering only bit values, making every generated question safe.

V. Classroom experience

CQG has been used in four different courses at the University of Victoria:

1) *CSc 111—Fundamentals of Programming wih Engineering Examples.*
2) *SEng 265—Software Development Methods.*
3) *SEng 360—Security Engineering.*
4) *CSc 361—Computer Communications and Networks.*

For *CSc 111* and *SEng 265*, there are eight quizzes based on the active code reading question types using C,covering the following topics:

1) *Loops.* The first quiz covers `for`, `while`, `break`, and `continue`.
2) *Parameter passing.* Many of the students have a Java background, where the language forces call-by-reference for objects and call-by-value for primitives. Consequently, they struggle with the use of `*` and `&`.
3) *Pointers and arrays.* Even for experienced programmers, there are misunderstandings on these topics.
4) *String libraries.* This quiz covers the use and sample implementations of the most popular functions.
5) *Linked lists.* The standard insertion, deletion, and search algorithms are covered.

In *CSc 111*, the quizzes are distributed across the term; In *SEng 265*, they are completed in the first four weeks.

For *SEng 265*, there are also four quizzes based on the active code reading question types using Python:

1) *Lists.* List creation, including list comprehension and list slices, are covered.
2) *Control structures.* The standard constructs are covered, focusing on looping over objects supporting iteration, such as lists.
3) *Functional features.* The `filter`, `map`, and `reduce` functions are covered, as well as lambda expressions.
4) *Regular expressions.* There are two quizzes on regular expressions, covering (1) simple matching and (2) string parsing.

In *SEng 265*, for example, the same approach scheme was used for each quiz. In each quiz, there were 20 questions; each question was selected randomly from a group with at least 25 questions. The quiz was made available for practice on Monday. Students took the quiz as many times as they liked. In the lab on Thursday the quiz was taken for a mark, with some additional questions and a 30-minute time limit. In the most recent offering of *SEng 265*, students were required

to develop their own *CQG* questions, focusing on n^2 sorting algorithms in C.

In *SEng 360*, there are five quizzes covering the Caesar, Substitution, Columnar Transposition, Vernam, Book, and RSA (with very small keys) encryption and decryption algorithms. Cryptanalysis using letter, digram and trigram frequencies is also covered.

In *CSc 361*, in addition to the *Hamming* question type shown in Figure 2(d), five question types were developed for the Fall 2012 offering:

1) *CRC*. The cyclic redundancy check procedure used for error detection in Ethernet packets is covered. This procedure uses a specialized binary long division.
2) *FDB*. The algorithm used to implement packet forwarding in Ethernet switches is covered.
3) *Dijkstra*. Knowledge of Dijkstra's shortest path algorithm is evaluated. No code is shown on the display. Instead, the student must provide the intermediate shortest path results for each iteration of the algorithm.
4) *IP address*. A pair of IP addresses and a subnet mask are provided; the student must determine whether the addresses are on the same subnet.
5) *NAT*. The algorithm used to implement the network address translation used in home routers is covered.

The *CSc 361* question types show the variety of procedural skills which can be evaluated using *CQG*. In five previous offerings, these procedures were presented in lecture and then evaluated on exams. In the Fall 2012 offering, after *CQG* practice, the students performed substantially better on the exams.

VI. Related work

Code Activator, a *CQG* predecessor, offered just the three active code reading applications: *Input/output*, *Bullseye*, and *Find-the-failure* [3], [4]. *CQG* improves on *Code Activator* in three important ways:

1) After using the *Code Activator* application, we realized the potential for a generalization: from individual product to product line. We significantly redesigned the system to make question type a plugin and implemented the six computer network question types.
2) We developed the question and the question type verification processes.
3) We used *CQG* in three more offerings of *SENG 265* and in *CSC 361*, developing many new questions.

The psychology literature contains reports that active retrieval is a more effective learning strategy than simple repetition [5]. Industrial software reviews have been shown to benefit from a question-based, active-learning approach [6]. Active learning has long been advocated in the Computer Science education community [7], [8]. A variety of approaches have been used to engage students. Liu has shown how to improve learning outcomes by replacing traditional lectures with a 50/50 split between lectures and question-based work sheets [9]. Wu's castle [10] is a graphical role-playing game for teaching students about the execution behaviour of loops. Students can set various loop parameters, e.g., index start and end values. To retain student interest, execution is carried out by animated characters. With PeerWise [11], a web-based tool for active learning, students collaboratively create multiple choice questions. Use of the tool correlates positively with later exam performance. Peer instruction [12], long used in physics instruction, was adapted to programming courses in Java. Clickers are used in lecture to answer questions focusing on previously assigned readings and exercises.

Some approaches to active learning incorporate automated checking of student work. GraphPad [13] is a web-based tool aimed at graph data structures. Students draw graphs which are compared to the instructor's solution automatically using a graph isomorphism algorithm. The tool also captures student interactions at the pen-stroke level for later study by the instructor. Kumar [14] presents a web-based tutor aimed at teaching C++ pointers. The tutor presents C++ code with a pointer error; the student must identify the variable and line number of the error. Both practice and marked mode are supported. The tutor is part of the problets framework [15].

With ProgTest [16], A student implements a program according to a specification. The program's correctness is automatically evaluated by running it against unit tests provided by the instructor.

VII. Conclusions

In lecture-based, online and MOOC courses, it is very useful to have online quizzes. *CQG* is a novel framework for developing online quizzes using template-based questions to evaluate procedural mastery. The *CGQ* framework includes detailed development processes for questions and for question types. Fourteen question types have been developed to date. *CQG* quizzes have been effective in evaluating procedural mastery in active code reading, encryption/decryption algorithms, and a variety of computer networking tasks.

Acknowledgment

The authors would like to thank H. Wang and H. Lien for help with authoring *C-doku* questions and the students of SEng 265 for their patience with bugs in the *C-doku* server.

References

[1] P. Carey, *New Perspectives on HTML, XHTML, and Dynamic HTML*, 4th ed. Cengage Learning, 2010.

[2] "The official web2py book," 2010, http://www.web2py.com/book.

[3] D. Hoffman and M. Lu, "A web tool for active code reading," in *Proc. FECS 2010*, 2010.

[4] D. Hoffman, M. Lu, and T. Pelton, "A web-based generation and delivery system for active code reading," in *Proc. SIGCSE*, 2011.

[5] E. Jaffe, "Will that be on the test?" *Assoc. Psych. Science*, vol. 21, no. 10, pp. 18–21, 2008.

[6] D. Parnas and D. Weiss, "Active design reviews," in *Proc. of the Intl. Conf. on Software Engineering*, 1985.

[7] J. McConnell, "Active learning and its use in Computer Science," in *Proc. SIGCSE/SIGCUE Conf. on Integrating Technology into Computer Science Education*, 1996.

[8] J. Whittington, "Increasing student retention and satisfaction in IT introductory programming courses using active learning," in *Proc. 2006 Informing Science and IT Education Joint Conf.*, 2006.

[9] Z. Liu, "Is paper-based work sheet out of date for CSII?" in *Proc. FECS 2009*, 2009.

[10] M. Eagle and T. Barnes, "Experimental evaluation of an educational game for improved learning in introductory computing," *SIGCSE Bull.*, vol. 41, no. 1, pp. 321–325, 2009.

[11] P. Denny, B. Hanks, and B. Simon, "Peerwise: replication study of a student-collaborative self-testing web service in a U.S. setting," in *SIGCSE '10: Proc. of the 41st ACM technical symposium on Computer Science education*, 2010, pp. 421–425.

[12] B. Simon, M. Kohanfars, L. Michael, J. Lee, K. Tamayo, and Q. Cutts, "Experience report: peer instruction in introductory computing," in *SIGCSE '10: Proc. of the 41st ACM technical symposium on Computer science education*. ACM, 2010, pp. 341–345.

[13] R. P. Pargas and S. Bryfczynski, "Using ink to expose students' thought processes in CS2/CS7," *SIGCSE Bull.*, vol. 41, no. 1, pp. 168–172, 2009.

[14] A. N. Kumar, "Data space animation for learning the semantics of C++ pointers," *SIGCSE Bull.*, vol. 41, no. 1, pp. 499–503, 2009.

[15] "Problets: the home page," 2010, http://www.problets.org.

[16] D. de Souza, J. Maldonado, and E. Barbosa, "Progtest: An environment for the submission and evaluation of programming assignments based on testing activities," in *Proc. CSEE&T*, 2011.