Designing Divergent Thinking, Creative Problem Solving Exams

Jeff Offutt
Department of Computer Science, George Mason University, USA
offutt@gmu.edu

Kesina Baral
Department of Computer Science, George Mason University, USA
kbaral4@gmu.edu

ABSTRACT

This experience report paper reports on case studies of final exams that used *creative problem solving* to assess students' ability to apply the material, and *divergent thinking* to ensure that each student's exam was unique. We report on two senior-level software engineering courses, taught a total of six times across three semesters. The paper gives enough information for readers to adapt this method in their own courses. Our exams include built-in divergence properties, which are elements or decisions that lead students to more than one good answer. Based on our experience, divergent thinking, creative problem solving exams not only provide excellent assessments of student knowledge, but ensure the integrity of the exam process, even when students work without supervision. We developed this strategy during the pandemic when all courses were online, but find them to be better than our traditional exams and are now using this strategy for in-person courses.

CCS CONCEPTS

• **Applied computing** → **Education**.

KEYWORDS

final exam, e-learning

ACM Reference Format:
Jeff Offutt and Kesina Baral. 2022. Designing Divergent Thinking, Creative Problem Solving Exams. In *44nd International Conference on Software Engineering:Software Engineering Education and Training (ICSE-SEET '22), May21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3510456.3514162

1 INTRODUCTION

This paper reports on experience with an unusual way to design and deliver final exams in software engineering courses. The concept was birthed by the needs of online teaching during the pandemic, and was successful enough to be used during our current in-person courses. Our experience covers two different courses and four semesters of use.

In March 2020, university teachers everywhere were subjected to an abrupt and unanticipated switch to online teaching. For most, this occurred mid-semester, well after the course had been designed. Most of us had little or no experience teaching or learning online, and were urgently scrambling to learn, adapt, and invent new pedagogies. Despite years of experience teaching thousands of university students, this experience crystallized the phrase "building the airplane while it is flying."

Our most urgent week-by-week need was to find ways to ensure students could keep learning–next week. But we also saw a large challenge looming down the road: How can we run a final exam that provides a valid assessment of our students' learning, fairly, equitably, and securely? One thing was immediately clear–we would not be the same room as our students during the exam.

We investigated lockdown browsers [8, 19], but identified many problems. They are prone to false negatives, because students can easily find many ways to cheat without being visible [9, 13, 18]. They are also prone to false positives, by flagging innocent behavior like fidgeting, staring out a window, or even being interrupted by pets and children This is particularly problematic for students with ADHD, haptic learners, and students with dark-toned skins [11, 12, 16]. Students also complain that lockdown browsers constitute an invasion of privacy [10], increase anxiety (already increased because of Covid), and they are problematic for students with technology limitations, including wifi problems.

The authors[1] decided to go into a different direction. The inspiration came from an international academic competition program called Odyssey of the Mind (OM) [6]. Offutt had coached an OM team of elementary students for several years, and had served as judge several times. OM poses intellectual problems to a team of students, who then design and implement solutions to those problems. For example, given a bag of different types of pasta noodles, some nails, and chewing gum, build a tower as tall as you can.

OM is considered an effective way to teach children to be future engineers and is supported by organizations such as NASA. OM features two major tenets that can be very helpful in ensuring the integrity of online exams.

First, Odyssey of the Mind emphasizes *creative problem solving*. Students are expected to come up with solutions that may or may not already be envisioned by the teachers. Second, OM requires students to exhibit *divergent thinking* in their solutions. A divergent problem has more than one solution, and divergent thinking means that different students will come up with different solutions. Some OM problems explicitly require divergent thinking, for example, given three or items, find as many *diverging* ways as possible to move a marble from one table to another.

Our key insight was that divergent thinking could ensure that each student provides unique solutions. This is easier in a problem-solving exam than with traditional questions, so we introduced divergent thinking in the context of a "*tech challenge*" exam. That is, we asked them to **apply** their knowledge in creative ways instead of **demonstrating** their knowledge by answering questions. The

[1] Offutt was instructor for all courses and Baral was a graduate assistant.

expectation is that every student will have a unique solution, and collaboration will be obvious to the instructor. The concept of divergent thinking had been found to be very useful in introductory programming courses [14] with programming assignments and in crowded classrooms with quizzes, but we had not tried the approach with a final exam before.

Our general concept started with dividing the exam into two parts. Part 1, worth one-third of the final exam score, was an open-book set of questions to check general knowledge. Part 2, worth two-thirds of the final exam score, was a tech challenge problem whose specifics depended on the course. It is important to note that the first time we used this format, we were completely unsure whether it would achieve our goals, including yielding a valid assessment while ensuring integrity of the exam. But if there is any good time to experiment with a creative, divergent thinking, teaching pedagogy, a pandemic panic semester when we all teetered on the edge of disaster seemed appropriate.

This paper starts with educational context of the reported approach in section 2. We then describe the goal, design and some sample divergent thinking exams in section 3. Findings, discussion, and assessment of our experience are described in section 4, followed by reflections in section 5. Conclusions are presented in section 6.

2 EDUCATIONAL CONTEXT

The overall context is important to any educational technique. Later in the paper, we discuss how the divergent thinking exam can be applied in other contexts, but here we describe how we used divergent thinking exams in two different software engineering courses across three semesters. Table 1 provides learning objectives and other details of the courses.

The software testing course starts with a module on software evolution, then teaches topics such as test automation, test driven development, and test criteria. It is required for all software engineering majors and is a senior elective for computer science majors. The web development course teaches usability concepts, then how to design and build web applications, primarily focusing on technologies such as J2EE [15] and React JS [7], and including concepts such as maintaining state on the web, handling concurrency and multiple users, and persisting data in databases. It is a senior elective for software engineering majors and for computer science majors.

Our experience includes six online sections with a total of 299 students, and one in-person section in Fall 2021 with 41 students. Enrollments are summarized in Table 2. All online meetings were taught synchronously.

Previous final exams were always in-person as a group, in a duration of 2 hours, 45 minutes. We call these "traditional" final exams in this paper. The traditional exams contained a mix of question types, including true-false, multiple-choice, short answer, term definitions, and problem solving. For example:

(1) `Faults and failures`
   `A Java method returned the wrong value, because a pair of parentheses was missing, causing the program to print the wrong number.`
   (a) From the above description, what was the fault and what was the failure?
   (b) Suppose the operation was "A + B * C" when it should have been "(A + B) * C". Give test values for the variables A, B, and C that would not cause the fault to result in a failure.
(2) `Draw a diagram to illustrate what happens when a JSP is accessed through a web browser`

3 DIVERGENT THINKING EXAMS

The key goal of a divergent thinking exam is that each student's answers should be different from all other students' answers. That is, students are expected to diverge from each other. **How** this is done depends on the course contents, the type of question, and the number of students. Larger courses may need more *divergence properties*, which we define to be an element or decision that allows for more than one correct answer. This section explores case studies of how we created divergence in our courses.

3.1 Goals of divergent thinking exams

We designed the exams to satisfy explicit goals. Sample final exam are available as a "reusability" package on github[2].

Goal 1: Exams should allow students to demonstrate that they achieved the learning objectives. This is an essential goal of all final exams, although it is possible that some LOs are assessed prior to the final exam.

Goal 2: The exam's design should ensure that each student's submission is different. This goal motivates our divergent thinking strategy. If each student's submission differs from other students' submissions, that goes a long way to ensure integrity of the exams. At the very least, it eliminates the possibility of two or more students developing their answers together and making cosmetic changes to pretend the submissions are different.

Goal 3: Exams should be challenging, but reasonable for well prepared students. While this goal probably applies to all exams, we want to acknowledge this need as an initial starting point. We were also concerned with our ability to estimate the time it takes students to solve problems, as well as variation among students.

Goal 4: It should be possible to grade exams in a reasonable amount of time. We considered "reasonable" to be "not significantly more" than a traditional exam for the same course. Time to grade is, of course, subject to many variables and challenging to measure. But it was also a significant source of concern. After all, each submission is different (by design!), meaning each submission would need to be evaluated separately and individually. Auto-grading was out of the question, as was pattern matching of given answers. And if a divergent thinking exam takes significantly more time to grade than a traditional exam, it would probably not be a viable approach.

Goal 5: It should be possible to create exams in a reasonable amount of time. This is similar to goal 4, and also quite difficult to measure.

Goal 6: The exams should separate students who learned the material at a high level as opposed to partial or minimal understanding.

[2]https://github.com/Keshina/divergentThinkingExam

Table 1: Course titles and learning objectives

Course	SWE 437	SWE 432
Title	Software Testing and Maintenance	Design and Imp. of Software for the Web
LOs	1. Knowledge of quantitative, technical, practical methods to test software 2. Testing techniques and criteria for all phases of software development 3. Knowledge of how to apply test criteria 4. Knowledge of modern challenges and procedures to update continuously evolving software 5. Understanding that maintainability and testability are more important than efficiency	1. Knowledge of engineering principles for designing usable web-based software interfaces 2. Understanding client-server and message-passing computing for web applications 3. Knowledge for building usable, secure, and effective web applications 4. Knowledge of how web applications store data 5. Knowledge of component software development technologies
Students	4th year, 25% software engineering, 75% computer science	4th year, 25% software engineering, 75% computer science
Prereqs	Data structures, discrete math	Data structures, discrete math

Table 2: Course sections taught

	SWE 437	SWE 432	modality
S 2020	60	43	partially online
F 2020		60	online
S 2021	68	68	online
F 2021	41		in-person
Total	169	171	

3.2 Part 1: Design of the open-book exam

Students were given the following instructions with the exams:

```
(1) You must answer these questions individually, without
    any help
(2) You may submit anytime between when the exam is distribu-
    ted and the day and time it is due
(3) You may submit answers as a text file, a PDF file, or a
    word document
(4) You must include your name in your answers
(5) Make sure that your responses are clearly labeled as to
    which question they answer
(6) You may not share this exam with anyone but yourself—doing
    so will be considered an honor code violation
```

Exams were distributed as editable word documents and students were encouraged to edit the document directly. Of course, we were fully aware that we could not enforce or check rule 1, but the exam was also take-home and open book, so the motivation for cheating was low. We also required students to sign the following statement:

```
I have not discussed this exam with anyone
except the instructor, I will not share the
exam with anyone else, and I will destroy
all copies, both paper and electronic.
```

We made a change in fall 2021. The exam was in-person, so students took part 1 in the classroom. The exam was still open-book and open-notes.

Our key divergence property for part 1 is to allow students to choose which questions to answer. We typically assigned 10 points to part 1, and gave 10 1-point questions. Each question had three choices, and students select which question to answer. An example terminology question from the testing exam is:

```
(1) Use one of the following terms to answer one of the
    three questions (1 point)
    {happy path tests, invalid input tests, minimum viable
    product, refactoring, spike, test suite}
  (a) What does it mean to go through a period of intense
      programming to "get ahead" of the TDD tests?
  (b) What is a type of tests that often are not designed
      when designing TDD tests?
  (c) What is changing the code to improve a non-functional
      quality, without changing the code's behavior?
```

An example conceptual writing question from the web development class is:

```
(1) Answer one of three (1 point)
  (a) Explain the difference between HTTP GET requests and
      HTTP POST requests.
  (b) What does separation of concerns mean?
  (c) Describe the benefits of a layered design pattern.
```

An example question about the technology Ajax is:

```
(1) Answer one of three (1 point)
  (a) What essential ability does Ajax interaction provide
      to web application developers?
  (b) How does Ajax affect security?
  (c) Explain the difference between authentication and
      authorization. How are they related?
```

Having each question be one point reduces grading effort. We tracked which of the three questions students answered in our spreadsheet, making it relatively simple to identify patterns and identify students who mostly answered the same questions. When students answered many of the same questions, we looked for similarities.

3.3 Part 2: Design of the problem solving exam

We called the problem solving part of our exams *tech challenge*, borrowing the term from a technique used in industry to select contractors. In both Spring 2020 courses, the entire exam was tech challenge. In subsequent semesters, part 1 was one-third of the exam and part 2 was two-thirds.

Here, students were given many choices to increase divergence. Our divergence properties were quite different in the two courses to reflect the different kinds of material. We describe them separately to help readers apply this concept to different courses. Later in the paper we discuss how divergent thinking exams can be used in other courses.

3.3.1 Software testing course. Students learned how to design tests by hand (human-based), how to model software artifacts and use test criteria to design tests (criteria-based), and how to automate tests using common tools such as JUnit and Selenium.

For the Spring 2020 tech challenge exam, they were given a web application that performed computations on numbers and strings. The web app had previously been used as an assignment in the web development course–we used the TA's solution for this exam. Students were given access to a running version[3] and three source Java files The files contained the front-end, the back-end, and a small enum object. The code-based tests (2 and 3 below) were based on the back-end module so that students did not need to understand how web apps work.

Students were asked to design three sets of tests through the following instructions (abbreviated here):

(1) **Input space partitioning (ISP)**: Model the external UI of the web app, develop an input domain model, then apply the base choice criterion [1] to design abstract tests. Transform the abstract tests into concrete tests (with input values), then run them by hand and report results.
(2) **Graph-based testing**: Draw the control flow graph for a specific method, annotate each edge to connect it with the source, then apply edge coverage (EC) to design tests.
(3) **Logic-based testing**: Draw the control flow graph for a different method, identify the logical predicates that control the execution of the method, then design tests to satisfy correlated active clause coverage (CACC) [1] (equivalent to masking MCDC [3]) on each predicate. Encode the tests in JUnit, then run the tests and capture the results.

At each step, the students made numerous design and presentation decisions. ISP requires design choices that results in many dozens of possible good input domain models, hundreds of values could be chosen, and they were free to choose how to represent their designs. Graph-based testing starts with a control flow graph and although the graph is unique to the method, any graph with more than a few nodes has dozens of isomorphic representations. They were also allowed to use any tool to draw the graph (including by hand), introducing more divergence. Hundreds of values could be chosen to satisfy each abstract test. Logic-based testing again starts with a graph, and during the semester students were encouraged to develop their own process for applying the criteria.

We used different programs in both Fall 2020 and Spring 2021. We also added more divergence by allowing students to choose from among two or three methods for graph-based testing and logic-based testing.

3.3.2 Web development course. Students learned how to design user interfaces for web applications (front-end), and how to design, develop, and deploy back-end software for web apps. The primary technologies were Java J2EE and ReactJS, although the course also introduced several other technologies and referenced a few more.

[3]https://cs.gmu.edu:8443/offutt/servlet/computeExample.compute

In Spring 2020, students were assigned a problem to manipulate boolean predicates. To support divergent thinking, this exam merged ideas from OM with software product families [5] by including two "minimal required elements (MRE)" that every student had to implement, and eight "optional required elements (ORE)," from which each student selected four to implement. The MREs were:

I. Your web app will accept a string that represents a boolean predicate that has boolean variables and logical operators, similar to 'if' statements in programs. Examples include: "**A OR B**"; "**x && y**"; "**M and N or Q**"; and "**today | tomorrow**". Note that you are expected to design the syntax and format, including the symbols used for the logical operators. For MRE credit, your web app only needs to handle ANDs and ORs, not parentheses, relational expressions ("x>0"), or other logical operators.
II. Your web app will process the string to create and display to the user a complete truth table for the predicate. "ANDs" and "ORs," using syntax you choose, are logical operators. Clauses in the predicate evaluate to boolean values, and are connected by logical operators, but do NOT include logical operators. Thus, the above 4 examples have 9 clauses in total: **A**, **B**, **x**, **y**, **M**, **N**, **Q**, **today**, **tomorrow**. The truth table for the first predicate, "**A OR B**," has 4 rows: true-true, true-false, false-true, and false-false.

As can be seen, several minor design and formatting decisions were left to the students as a way to increase divergence. We included the note: "*Note that you are to design the input and screens, design how the truth tables are displayed, and design and write code to find clauses within the predicates. I am intentionally not giving you requirements about how the UI will look or behave.*"

The MREs were worth 60% of the final exam score. Students chose 4 of 8 OREs, at 10% per ORE, to complete their grade. The OREs were:

(1) Input validation–report invalid strings.
(2) Allow parentheses in the predicates, for example, "**((A or B) and C)**".
(3) Evaluate the predicate for each combination of truth values. For example, for the predicate "**A OR B**," the result values would be true, true, true, false; using the same order as the rows in MRE #II above.
(4) Add an additional screen, after the truth table is displayed, to evaluate specific truth-values. The user can assign truth-values to the clauses, and your app will return the value of the predicate. Your software will need to maintain state through the multiple requests (any method is allowed).
(5) Include an additional logical operator–exclusive or.
(6) Allow multiple syntaxes in the input box for logical operators (for example, "&", "&&", "AND", "and").
(7) Allow constant binary values to be included in the predicate (for example "**((A and (TRUE or C))**").
(8) Give the user the option to display the truth-values in the truth table in different formats, such as "t-f," "T-F," "1-0," "true-false," etc.

Both authors took the exam early, resulting in several refinements. We found that one small part took at least an hour and a half (a recursive algorithm that was not related to the course material), so we provided the algorithm on the exam. It is a challenging algorithm to design, but was also available on the web, and since that was not the subject of the class, we saw no value in penalizing students who did not find the algorithm online and struggled to design it by hand.

This tech challenge exam had many divergence properties. Students were allowed to use any technologies in their implementation. They could deploy on github-heroku, AWS, or elsewhere. They were required to submit a URL to a running program, and all source files, including .js and .css files. Each requirement required design decisions that could lead to multiple good answers. They chose 4 of 8 OREs, which allowed as many as 70 possible combinations. (In practice, nobody understood or chose ORE #4, so it was really only 35 combinations.)

After our first semester's experience, discussed later in the paper, we created smaller and simpler problems for Fall 2020 and Spring 2021.

The Fall 2020 exam asked students to sort strings. The three MREs asked students to (I) accept a list of strings and return them in sorted order, (II) sort the strings in either ascending or descending order, and (III) allow an unlimited list of strings. We only had 5 OREs and students selected 2:

(1) **Numeric**: The user can choose to sort in numeric order (assuming the strings are actually numbers).
(2) **Sanitize**: Dangerous strings such as "<script>", "<javascript>", and "onLoad" are removed before sorting.
(3) **Unique**: If the list has two identical strings, only one will be returned in the sorted result. For example, if the submitted list is [shirt, pants, shirt, shoes], the web app will return the list [pants, shirt, shoes].
(4) **Forward**: Your backend uses the dispatcher.forward() method to forward the request from one servlet to another.
(5) **Alternate sorts**: The user can choose to sort based on an additional quality, such as string length or character position. Be sure to make it clear what the alternate sort is.

The Spring 2021 exam ask students to build a web app to compute averages. The MREs were (I) accept a list of integer values and return the average, (II) the user can choose mean, median, or mode, (III) the number of integers is not limited. For OREs, they could (1) add standard deviation, (2) add double (non-integer) values, (3) remove duplicates, (4) sanitize the inputs, or (5) use the J2EE forward mechanism.

4 FINDINGS, DISCUSSION, AND ASSESSMENT OF EXPERIENCE

This section presents findings and observations from giving six different divergent thinking final exams across four semesters. The answers were somewhat different in different semesters as we learned and evolved the strategy.

4.1 Findings from the testing class

We gave divergent thinking final exams in Spring 2020 to 60 students (second half was online), in Spring 2021 to 68 students (completely online), and Fall 2021 to 41 students (completely in-person). The format and procedure was slightly different in each semester. In Spring 2020, the format of the exam was entirely tech challenge, creative problem solving. In Spring 2021, we added part 1 of the exam to assess and reinforce general knowledge, and gave part 1 in the classroom in Fall 2021.

In our first exam in Spring 2020, we presented the exam online at the beginning of the designated final exam period, and the students were told to remain in the zoom session until they completed.

In our second exam in Spring 2021, we moved the tech-challenge portion to a 24 hour format. Part 1 was given at the beginning of final exam week, and students submitted anytime between then and the beginning of the designated exam period. The class met synchronously at the beginning of the designated final exam period, and the tech-challenge portion of the exam was shared. The instructor stayed online to answer questions throughout the period (2 hours, 45 minutes), but students were allowed to submit anytime during the following 24 hours. We did this to allow for technological failures and as a general effort to reduce stress.

Goal 1 was that the exam should allow students to demonstrate that they achieved the learning objectives. The problem-solving, tech challenge, nature of the exam meant that it focused more on skills (modeling and test generation) than on the abstract and theoretical concepts. But since they had to start with source, then create an abstract model, then design tests, and then automate those tests, the exam was able to assess learning objectives 1, 2, and 3 from Table 1.

Goal 2 was that the exam's design should ensure that each student's submission is different. The many choices and the amount of design needed, including modeling input domains, drawing graphs, choosing specific input values, and the structure of the automated tests made it easy to see that all submissions were clearly unique. Instead of a single answer, tens of thousands of good solutions could have satisfied the requirements of this exam.

Goal 3 was that exams should be challenging, but reasonable for well prepared students. In Spring 2020, almost all students were able to complete all or most of the exam within the 2 hours and 45 minutes allotted. The scores were high (average 91%) but students were clearly challenged on several parts. The time to complete ranged from 75 minutes to 3 hours, and fewer than 10 students used a 15 minute grace period. There were two outliers, both of whom had technical problems during the exam period. One student's computer crashed during the exam and another student lost electricity. We granted both 24 hours to complete, an ad-hoc adaptation that was codified the following semester.

A few students complained about stress from writing JUnit tests (which requires some programming), but most did fine. Even students who vented on the chat about the JUnit were able to finish on time. One sent the following message before dropping off: "Prof, thanks for letting me '*complain*,' that helped me calm down." One student who was not able to complete the JUnit tests admitted to having allowed his partners to do all the JUnit programming through the semester. That's a good lesson about the impact of collaborative assignments on student performance, although perhaps not related to the format of this exam.

In Spring and Fall 2021, student performance was similar (averages of 87% and 88%), and most students reported spending less than two hours during the 24 hour period.

The choice of the program to test was crucial to goal 3. Students created system tests and unit tests, so the software had a simple UI but with enough choices to lead to variations in how to model it. In both semesters, we had them test a small program (one Java class with a few hundred lines of code) that was deployed as a web application so they could easily run it. Both programs had at least one method whose control flow had enough branches to test their knowledge of graph-based testing, and another method with a predicate complicated enough to ensure the logic testing was non-trivial. They wrote automated tests for a method that modified class level variables that needed to be checked in assertions, which are often overlooked. More than half of the students did that correctly, even though this is a very common mistake in industry [2].

Goal 4 was that it should be possible to grade exams in a reasonable amount of time. All exams in all semesters were graded by Offutt. This was the big surprise of the divergent thinking exam. We assumed grading exams where each submission was different would take more time than grading traditional exams; hopefully not too much more time. We completed the entire exam as preparation, doing all optional parts. With this preparation, the grading was not only surprisingly fast, it was **faster** than grading traditional exams! It took on average about 10 minutes per exam. After 5 or 10 exams, we were able to construct a short list of common mistakes, which increased the speed. As a comparison, we have found that traditional exams for this course usually take about 30 minutes to grade per exam.

Goal 5 was that it should be possible to create exams in a reasonable amount of time. Evaluating this goal is quite complex, because the process was completely different. A common process with a traditional exam is to review all class materials, formulate questions based on the knowledge and concepts, then refine and format those questions for the exam. A separate pass checks for consistency and coverage of the learning objectives. This takes several hours and is usually done between the last class meeting and the final exam. Creating a divergent thinking, problem solving, exam is completely different. The first step is to find or invent a program that students can understand and analyze in a reasonable amount of time, and that has enough features to demonstrate the knowledge of testing. This is a very creative step that could take very different amounts of time for different instructors and different topics. We found that reusing assignments in one class in another class's final exam was very helpful. Overall, designing divergent thinking exams took less time for us, but your mileage may vary.

Goal 6 was that the exams should separate students who learned the material at a high level as opposed to partial or minimal understanding. We compared final exam scores with quiz and assignment scores, and they were generally consistent. Most mistakes were clear and usually easy to connect to gaps in students' knowledge. By asking them to demonstrate their knowledge by designing tests, we were going beyond memory tricks, and checking both the "apply" and the "create" levels of Bloom's taxonomy [17]. After grading the exams, we were confident that we were effective at determining who is able to model software, design tests, and implement those tests well.

4.2 Findings from the web development class

We gave divergent thinking final exams in three sections of the web development class, in Spring 2020 to 43 students, in Fall 2020 to 60 students, and in Spring 2021 to 68 students. As with the testing exam, Spring 2020 was entirely tech challenge exam, and a question part was added in Fall 2020 and Spring 2021.

As in the testing course, in Spring 2020, we presented the exam online at the beginning of the designated final exam period, and the students were told to remain in the zoom session until they completed. In Fall 2020 and Spring 2021 we followed the same schedule as with the testing exam.

Goal 1 was that the exam should allow students to demonstrate that they achieved the learning objectives. As with the testing exam, this exam assessed practical skills (they built and deployed a complete web app!), but did not assess deep knowledge of theory or abstract concepts. This covers LOs 1-4 from Table 1, and partially LO #5.

Goal 2 was that the exam's design should ensure that each student's submission is different. This goal was met exceedingly well. The students designed different UIs, used different technologies (mostly J2EE and ReactJS), implemented different sets of features, and had very different designs. We could not rule out the possibility of students helping each other debug, but we're not sure that matters. It was very clear that everyone did something completely different.

As an example of the diversity, in Spring 2020, 33 students used J2EE (servlets), 8 used ReactJS, 3 used other JS frameworks (including NodeJS and Angular), 9 used plain JavaScript, and one used PhP (which was not taught in the course). Some students used multiple technologies. The Spring 2020 exam had eight optional elements and students selected four. Only a few selected the same four, and only one combination was selected by more than three students.

Goal 3 was that exams should be challenging, but reasonable for well prepared students. In Spring 2020, the time to complete varied greatly, much more than the testing exam. This exam was mostly programming, and it is likely we discovered, just like many before us, that different people need dramatically different amounts of time to write the same program. More than half the students finished between 60 and 120 minutes, and most of the rest finished within the allotted 2 hours 45 minutes. Yet some took many hours, and we made an ad-hoc decision to not enforce the deadline. Six students took more than 8 hours to complete. We also saw almost no correlation between time to complete and quality of the program. Of the last three submissions, one was in the top 10%, one was in the middle, and one was near the bottom. As a comparison, the exams with the two lowest grades were submitted in under two hours.

As a result of these observations, in Fall 2020 and Spring 2021 we designed simpler problems and explicitly allowed 24 hours to complete. This not only allowed for the divergence in programming speed, but also reduced stress.

Goal 4 was that it should be possible to grade exams in a reasonable amount of time. As with the testing exam, we were very pleasantly surprised at the ease of grading. Every UI was different and students implemented different features, so using automated tests to grade was not possible. Instead, we created sets of standard tests in a spreadsheet, then entered the relevant values by hand. Each exam took about 10 minutes to grade, including a review of the design and quality of code. The only exceptions were a few students whose deployment failed (we used github with Heroku). Most fixed the problem within a few minutes and only one was not able to get their program to work.

Goal 5 was that it should be possible to create exams in a reasonable amount of time. Most of the comments from subsection 4.1 are the same for this course. The process is even more creative, as the instructor is creating a separate project. However, the idea generation can happen at any time, including before the class starts. Creating the actual exam only takes a few minutes, as most of the text is the same each semester.

Goal 6 was that the exams should separate students who learned the material at a high level as opposed to partial or minimal understanding. We learned who could design and build small web applications. However, the course taught small and subtle concepts that did not show up in this tech challenge. Most were covered in quizzes throughout the semester, but if we had tried to embed them in this challenge, the project would have been bigger and harder to finish.

5 REFLECTIONS

This section reflects on our findings from the case studies, and includes opinions that are based on our experience. The most surprising finding was that these exams took less time to grade than traditional exams. While grading, the goals and expectations were clear and evaluating the results took less judgement than with traditional exams. Very importantly, the divergent thinking exams included very little writing. We are not English teachers and our students are not writing students, so it is sometimes a challenge to understand our students' free form text answers.

A more interesting conclusion is that, in our judgement, divergent thinking exams are better at assessing students' knowledge of the material. More than anything else, software engineering students are learning how to build high quality software. Assessing conceptual and theoretical knowledge is fine, but what really matters is whether these students can join a software company and apply their knowledge. A creative problem solving exam assesses that ability to apply, and a divergent thinking exam ensures that each student's work is unique.

We next consider weaknesses of divergent thinking exams. As to integrity of the exams, we have to recognize that all work was unsupervised. It is possible that a student could have someone else do the entire exam. We believe that risk is low and are willing to accept it.

Another loss is that we have traditionally used challenging questions to separate the top few students from the really good students. We usually ask questions about theoretical knowledge and do not allow students to look the answers up. Four semesters of divergent thinking exams has made us question how much this matters. While PhD programs may need to differentiate "the best" from "the good" student applications, industry usually does not. In most organizations, only the software architects need to know the deep theory and its application–and companies do not hire entry-level new graduates to be software architects.

6 CONCLUSIONS

Based on the positive experience through online teaching, we elected to continue with divergent thinking exams for an in-person class in Fall 2021. The results were just as positive and, in our judgement, the exams were just as effective both as assessment devices and as learning devices as traditional exams were.

This paper presents the model of a divergent thinking, creative problem solving exam, and explores how we applied it to two quite different courses. Creating divergence properties would be different for other courses. To generalize our approach, we sought elements of the exams where teachers could made decisions when formulating the questions, and pushed the decisions to the students. This works particularly well for design decisions, presentation choices, and cosmetic options. For example, in a lower-division programming course, students could be given a partial program, then asked to select one or two out of a number of features (similar to the OREs of the web development course). With larger classes, we simply need more choices or more divergent properties. Divergent thinking is a powerful and general concept that can be applied in many situations.

A basic tenet of Universal Design for Learning (UDL) [4] is that accommodations made for disadvantaged people are almost invariably good for non-disadvantaged people. We found that to be true here. We made massive changes to our traditional final exams to accommodate our online learning approach where we were all disadvantaged, and concluded that these change are good for all. From now on, divergent thinking, tech challenge, creative problem solving exams is our new traditional model.

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2017. *Introduction to Software Testing* (2nd ed.). Cambridge University Press, Cambridge, UK. ISBN 978-1107172012.

[2] Kesina Baral and Jeff Offutt. 2020. An Empirical Analysis of Blind Tests. In *13th IEEE International Conference on Software Testing, Validation, and Verification (ICST)*. IEEE Computer Society, Porto, Portugal, 254–262.

[3] John Chilenski and L. A. Richey. 1997. *Definition for a Masking Form of Modified Condition Decision Coverage (MCDC)*. Technical Report. Boeing, Seattle, WA. http://www.boeing.com/nosearch/mcdc/.

[4] D. L. Edyburn. 2013. Critical issues in advancing the special education technology evidence base. *Exceptional Children* 80, 1 (2013), 7–24. https://doi.org/10.1177/001440291308000107

[5] Hassan Gomaa. 2005. *Designing Software Product Lines with UML*. Addison Wesley Object Technology Series, Boston, MA.

[6] Creative Competitions Inc. 2021. Odyssey of the Mind. Online. https://www.odysseyofthemind.com/, last access October 2021.

[7] Facebook Inc. 2021. React. Online. https://reactjs.org/, last access October 2021.

[8] Respondus Inc. 2021. Respondus. Online. https://web.respondus.com/he/lockdownbrowser/, last access October 2021.

[9] Jessica Kasen. 2021. Gradebees. Online. https://gradebees.com/cheat-online-tests/, last access October 2021.

[10] Eleni Kopsaftis. 2020. Over 4700 signatures against the LockDown Browser at U of G. Online. https://theontarion.com/2020/12/04/over-4700-signatures-against-the-lockdown-browser-at-u-of-g/, last access October 2021.

[11] Hugo Smith Maddy Andersen. 2020. "Essentially Malware": Experts Raise Concerns about Stuyvesant's Lockdown Software. Online. https://www.stuyspec.com/quaranzine/essentially-malware-experts-raise-concerns-about-stuyvesant-s-lockdown-software, last access October 2021.

[12] Callie McNorton. 2020. LockDown Browser is an invading privacy. Online. https://georgiastatesignal.com/lockdown-browser-is-an-invading-privacy/, last access October 2021.

[13] Sean Miller. 2020. Lockdown Browser is bad software and should be scrapped. Online. https://mytjnow.com/2020/12/02/lockdown-browser-is-bad-software-and-should-be-scrapped/, last access October 2021.

[14] Jeff Offutt, Paul Ammann, Kinga Dobolyi, Chris Kauffman, Jaime Lester, Upsorn Praphamontripong, Huzefa Rangwala, Sanjeev Setia, Pearl Wang, and Liz White.

2017. A Novel Self-Paced Model for Teaching CS1 and CS2. In *Learning at Scale*. ACM Press, Boston, USA, 1–4.

[15] Oracle. 2021. Java 2 Platform, Enterprise Edition (J2EE) Overview. Online. https://www.oracle.com/java/technologies/appmodel.html, last access October 2021.

[16] Vivian Pham. 2021. Lockdown browsers fail to create a culture of academic integrity. Online. https://retriever.umbc.edu/2021/04/lockdown-browsers-fail-to-create-a-culture-of-academic-integrity-they-invade-student-privacy-and-harm-student-health/, last access October 2021.

[17] Terry Scott. 2003. Bloom's taxonomy applied to testing in computer science classes. *Journal of Journal of Computing Sciences in Colleges* 19, 10 (October 2003), 267–274.

[18] Superamaz. 2021. How to cheat on Respondus lockdown browser without getting caught. Online. https://amazfeed.com/how-to-cheat-on-respondus-lockdown-browser-without-getting-caught/.

[19] webassign/cengage. 2021. Webassign. Online. https://webassign.com/instructors/features/secure-testing/lockdown-browser/, last access October 2021.