published as: Thomas Staubitz, Ralf Teusner, Christoph Meinel, Nishanth Prakash: Cellular Automata as an Example for Advanced Beginners' Level Coding Exercises in a MOOC on Test Driven Development in International Journal: Engineering Pedagogy (iJEP), vol. 2/2017

Cellular Automata as an Example for Advanced Beginners' Level Coding Exercises in a MOOC on Test Driven Development

Lessons Learned and Suggestions for Improvement

https://doi.org/10.3991/ijep.v7i2.6969

Thomas Staubitz, Ralf Teusner, Christoph Meinel
Hasso Plattner Institute, Potsdam, Germany
{firstname.lastname}@hpi.de

Nishanth Prakash
Brown University, Providence, RI, USA
nishanth_prakash@brown.edu

Abstract— Programming tasks are an important part of teaching computer programming as they foster students to develop essential programming skills and techniques through practice. The design of educational problems plays a crucial role in the extent to which the experiential knowledge is imparted to the learner both in terms of quality and quantity. Badly designed tasks have been known to put-off students from practicing programming. Hence, there is a need for carefully designed problems. Cellular Automata programming lends itself as a very suitable candidate among problems designed for programming practice. In this paper, we describe how various types of problems can be designed using concepts from Cellular Automata and discuss the features which make them good practice problems with regard to instructional pedagogy. We also present a case study on a Cellular Automata programming exercise used in a MOOC on Test Driven Development using JUnit, and discuss the automated evaluation of code submissions and the feedback about the reception of this exercise by participants in this course. Finally, we suggest two ideas to facilitate an easier approach of creating such programming exercises.

Key Words— programming tasks; unit testing; test driven development; MOOC; automated grading

1 Introduction

The well-known Game of Life [7], a John Conway creation, has withstood the test of time among popular programming problems for good reason. It exercises several

useful programming concepts (like the use of random numbers, 2-D arrays, functions, loops and recursion), can be extended to include advanced programming concepts and methodology (such as Object Oriented Programming, Design Patterns, Test Driven Development, UI design, Interactive Programming, etc.) and can be scaled to varying levels of difficulty with ease. While some problems in Cellular Automata and Game of Life, are well within the reach of programming novices, others are difficult enough to be considered for advanced programmers in specialized areas (such as parallel programming or ecology modelling, etc.). Most importantly it is fun and engaging to program Cellular Automata and watch their transitions. In this paper, we explore the suitability of Cellular Automata challenges for developing a wide variety of programming assignments, appropriate for audiences ranging from novices to experts. Programming ability basically rests on two pillars. One is formed by the theoretical foundations of knowledge about computers, programming languages, tools and formal methods (i.e. knowledge which is of a declarative nature, for example— being able to state how a "for" loop works). The other is the ability to apply this knowledge hands-on.

The famous adage "I hear and I forget. I see and I remember. I do and I understand", as succinctly put by Confucius, is very much relevant even to this day in pedagogy. But while much of the instructional content focuses on theoretical aspects of computer programming, it is programming assignments and exercises that play a major role in helping students learn programming skills and techniques through practice.

Providing students with quality homework, exercises, and assignments is integral to the success of any course. Particularly so, in introductory courses where the major share of student's learnings comes from [1]. Often however, the tasks set for practice are not considered as a vehicle that can direct learning behaviors in students [2]. To add to this, most programming assignments are what most students would classify as boring: mathematical problems (except games), sorting, string manipulation and others all succeed in helping students to learn the concepts, but few are met with any real enthusiasm and fewer still inspire true creativity [3].

In Section 3 we begin with a study of interesting programming tasks that have been previously used in classroom courses, MOOCs and other instructional settings. In Section 4 we then provide a brief introduction to Cellular Automata, their different types and modelling use cases, and in Section 5 we describe how they can be used as programming tasks in various instructional contexts. We discuss previous work related to this in Section 2 and in Section 6 we present a study on the use of Cellular Automata programming as an exercise task in the context of a MOOC on Test Driven Development using JUnit. We also discuss student submissions evaluation methodology in this exercise and present an analysis on student feedback and reception. In Section 7 we present the next steps that we've already started to implement.

The main contribution of this paper is to explore the various types of problems and the varying levels of complexity and difficulty, that can be constructed using concepts from Cellular Automata and the suggestions how to lower the barrier for teaching teams to provide practical programming exercises to their courses.

2 Related Work

There has been extensive research in Cellular Automata, and the problem of designing programming tasks in computer science pedagogy. However, one of the results of our literature review was that there has been surprisingly little work investigating the utility of Cellular Automata programming problems in computer science education.

Using a series of Cellular Automata modelling examples, Lilly shows how these problems can be used creatively to address problems in the areas of motivation, learning styles, development of modeling skills, and the teaching of technology [5]. While the authors do describe the use of Cellular Automata in classrooms, they do not discuss it from the perspective of computer science pedagogy.

Weeden employs multiple versions of the Game of Life simulation as exercises in parallel programming, including exercises using shared or distributed memory as well as exercises on how to measure the performance and scaling of a parallel application in multicore and many-core environments in [16]. Mache and Karavanic describe their use of the Game of Life as an exercise in teaching parallelism by asking students to speed up a CPU-only implementation by modifying it to use CUDA [17]. Wick employs the Game of Life as a vehicle to teach freshman students Command and Visitor, two important and widely applicable design patterns, by refactoring the Game of Life application [18]. Furthermore, Beniak uses the Game of Life to teach principles of game design and game engine development with Microsoft XNA [19]. Although each of these endeavors explore interesting cases of cellular automata problems used in computer science education, we do not find any holistic discussion on the possibilities in using cellular automata for programming tasks.

3 Programming Tasks

While course lectures impart theoretical knowledge to students, programming exercises and tasks set in the course have the bonus of complementing the lectures by imparting practical knowledge to students. They help students in gaining a deeper understanding of the subject and in enabling them to apply their knowledge in new situations [4].

Guzdial and Soloway propose that this is achieved best by applying tasks in the realm of media. They claim that the current generation[1] of students enjoys learning about array manipulation better if the example results in producing sound more than if the task requires sorting student IDs or doing linear searches for employee names [4]. This calls for better use of current technology in pedagogy, where more immersive

[1] We dare to suggest that previous generations of students might have preferred that as well. Nowadays however, the improvements in technology easily allow to do that.

experiential learning can now be easily created by including manipulation of sound, graphics, and videos in problem contexts.

Feldman and Zelenski suggest that the tasks that are suited best, are those that result in programs that students want to write for the reason that they enjoy running them themselves. Apart from requiring a strong audio-visual component and a high degree of interactivity, they believe that the end result of an assignment must be worth the time and effort required to achieve it, because when students see the end result of a programming assignment as something especially impressive, useful, or fun—a program they would like to have for themselves—they will approach the project with a heightened sense of interest and motivation. Also, writing a programs that can be presented with pride to relatives and friends, significantly increase a beginning programmers sense of accomplishment. [1]

An obvious class of problems that fits the above requirements very well are those of game programming. Very often, the students say that gaming is what got them interested in computers and regardless of where they end up in their careers, many start off with a desire to become game designers. The fun factor in games is what sets tasks in this context apart from the majority of problems assigned. Next to this, games can also hold the potential for integration of almost all of the concepts and techniques taught in a typical CS degree program [3].

Although the Game of Life and other Cellular Automata are not games in the conventional sense, they are found to be equally engaging. There are typically no players, and the game is generally not about winning or losing, but is typically used as a simulation of another system, that runs according to some specified rules. Thus, they combine the benefits of gaming with the benefits and challenges of mathematical exercises.

4 Cellular Automata

A cellular automaton is a mathematical model which has been widely studied in the simulation of various physical, chemical and biological systems. They usually consist of a configuration of "cells" which represent elements of the system being modeled, each of which can be said to be in one of a set of finite number of states. The configuration of these cells can be a single row of cells as in one-dimensional or elementary cellular automata, a grid as in two-dimensional cellular automata, blocks placed in three-dimensional space as in three-dimensional cellular automata or other regular structures such as a grid of hexagonal cells, etc. Each cell has a defined neighborhood, generally depending on the shape and configuration of the cells. The cell itself may or may not be included in the neighborhood. If the cells are in a row, a cell has two neighbors—left and right. If the cells are in a hexagonal grid, a cell has six neighbors. The cells that are located on the margins may have their outer neighbors either defined as dead or as the cells that correspondingly lie on the opposite end of the configuration. The cellular automaton starts with an initial combination of states of its cells and evolves following a transition function (a set of rules) that define the next states of the cells depending on the current states of their

neighbors. The definition of neighbor and the transition function can vary and be complex depending on the system being modeled.

For example, the cellular automaton could be modeling microbial growth, where each cell represents a microbe cell which can be alive or dead, and the rules by which it evolves could be that it survives or is born if there are four or more live adjacent cells, otherwise it dies. The previously mentioned Game of Life is modeled similar to this. Also, a simple two-dimensional Cellular Automaton can model growth of crystals or patterns in snowflakes or on shells. These sequences of transitions are both mathematically interesting as well as aesthetically pleasing when displayed using colors to represent states of the cell [6].

Types of Cellular Automata vary widely in their complexity and modelling ability. While some models can only be used to express a basic idea of a phenomenon, others are accurate enough to be used for prediction. Stephen Wolfram describes this in [25] as:

"Cellular automata are sufficiently simple to allow detailed mathematical analysis, yet sufficiently complex to exhibit a wide variety of complicated phenomena."

Even simple Cellular Automata, such as the Game of Life, are computationally universal, meaning that it is able to compute/model anything computable [12, 24]. From the spots on a leopard to the design of a snowflake to the structure of the human brain, Wolfram is confident that there is a cellular automaton that encodes the design of each [10]. This nature of complex phenomenon emerging from simple systems in Cellular Automata [34], has instigated several scholars to consider the question of whether the underlying model of the universe is a cellular automata populated by digital particles [35, 36]. In the usage of Cellular Automata concepts for programming problems, this variety gives us the ability to tweak difficulty and complexity to suit our needs, to weave interesting concepts together to make an engaging experience for the problem solver, and at the same time stay relevant to topics in Computer Science curriculum.

5 Relevance of Cellular Automata in programming tasks

Jon Conway popularized cellular automata through The Game of Life [12], and Martin Gardner made them reach the public through his columns in Scientific American [7, 8] and his puzzle collection books [9]. Since then, a great number of professional mathematicians, as well as amateurs have contributed to an understanding of the game of life [12, 13, 14, 15], as well as Cellular Automata [20, 21, 22, 23]. Due to their engaging and narrative nature they have also been adopted widely to teach programming concepts.

Apart from their popularity and engaging nature, the most important feature of Cellular Automata problems is the fine grain control they provide to the teacher, in being able to tweak the difficulty of problems by making incremental enhancements to the problem design. For example, if the problems required to program an Elementary Cellular Automaton following a specific rule, the next difficult problem

could require the programming of an Elementary Cellular Automaton using only one array, thus imparting list processing skills to problems solvers. The next addition could be that a general Elementary Cellular Automata generator has to be coded, taking the rule number (a naming convention that maps to a unique transition function) as parameter. The following task could then be to code the Game of Life or any other two-dimensional cellular Automaton using two dimensional arrays which can be scaled then to 3 dimensions, etc. Interesting variations that stimulate one's visualizing ability, could require the Cellular Automaton to wrap around at its edges to resemble a Torus or only to wrap around on the sides as a Mobius strip. For more algorithmic variety and difficulty ranges, problems could require cells of other, non-rectangular shapes such as hexagons; or define unidirectional neighbors or have other such complex definitions of neighborhood.

To include probability concepts, the task can be to design stochastic cellular automata where the transition rules are probabilistic rather than definite (i.e. instead of stating that the cell would be dead in the next state, we say the cell has 80% chance of dying). The states of the cells could be continuous rather than discrete as in Continuous Cellular Automata, which tends to model many Finite Element Analysis implementations [37]. Furthermore, the automaton can be required to have a continuum of locations as in Continuous Spatial Automata or have time as a continuous variable where the state evolves according to differential equations, thus integrating important concepts from Calculus.

The second most important feature is that they provide teachers with the ability to easily integrate a wide variety of programming concepts into the problem. The simpler variants of Elementary Cellular Automata suit best in the procedural programming context. To teach the concepts of Object Oriented Programming (OOP), variations from the simpler cellular automata can be employed. These variations might range from requiring the cells to be movable within the universe of the cellular automaton, to requiring the possibility to nest cells in another cell, thus enabling the student to model an entire ecosystem. Due to its parallelizability, the game can also be coded in Functional Programming paradigms which are most well suited for parallel programming. The variety in the spectrum of Cellular Automata problems recommends them to motivate students. Particularly, visual learners are attracted by the created patterns and encouraged to develop their modeling skills [5]. They are considered to be useful for the development of curricula to teach certain computer technologies [5]. Problems of varying depth can be employed to expose different approaches to solve a task. The arising difficulties can be employed as feedback to improve the teaching material.

Recent investigations by Stephen Wolfram [11] on cellular automata have put forth multiple thought provoking questions on the nature of our universe, on computability and computational irreducibility, and on the epistemology of sciences. His extensive research in these areas have exposed important unanswered question related to theoretical computer science, logic, Artificial Intelligence, Mathematics and Philosophy. Deep questions such as these can generate sustained interest among some students that may lead them eventually to take up a career in Computer Science research.

6 Case Study (Use of Rule 54 Elementary Cellular Automata Programming Task in Test Driven Development using JUnit MOOC)

In the following we will discuss the findings of a case study that we conducted during our MOOC "Introduction to Test Driven Development in Java and JUnit." The course was designed as a two-week workshop on the basics of Test-driven Development. The target group were participants with basic Java knowledge. About half of the participants had also participated in the previous Java programming course that we had offered a year earlier. The majority of the participants considered themselves to have good to excellent knowledge in programming. A couple of questions that we have asked to double-check these self-evaluations seem to confirm this. An in-depth examination of this survey will follow in a future paper. The course had 2799 registered participants[2]. 950 of these never showed up[3]. 283 participants received a Record of Achievement. 322 of the participants answered a couple of questions in our course end survey. The age of the participants ranged from less than 20 to 69 years. Most of the participants were male, and not surprisingly, as the course was offered in German, lived in Germany. The overall feedback on course quality, course length, etc. was good to very good. The difficulty of the course was considered medium (3 on a 5 point Likert scale ranging from 1-very easy to 5-very difficult.)

We picked the Rule54 Automaton among the one-dimensional automata rules, as it was the best fit for our requirement of being easy to comprehend and, therefore, being suitable for beginners. Any other rule would have been possible as well, coming along with its own advantages or disadvantages.

6.1 Rule 54 CA

A rather simple form of cellular automaton is the one-dimensional (elementary) cellular automaton, which consists of a single row of cells. Each cell starts with a given initial state and evolves depending on the states of its left and right neighbor.

In our course on Test-driven Development and JUnit, we provided an exercise based on an Elementary Cellular Automaton called *Rule 54*. The participants were asked to implement and test this Cellular Automaton, which is constituted by the following set of rules:

1. If the cell and both neighbors are dead in the current state, then the cell is dead in the next state.
2. If the cell and both neighbors are currently alive, then the cell is dead in the next state.

---

[2] We always employ the number of enrollments at course middle as the basis for our calculations, as these are the participants who still have a realistic chance on finishing the course with a certificate

[3] These are what we call no-shows. Platform users that register for a course but in the end do not ever visit a single item of the course.

3. If the cell and one of its neighbors are currently alive, then the cell is dead in the next state.
4. If the cell is alive and both neighbors are dead in the current state, then the cell is alive in the next state.
5. If the cell is dead and at least one of the neighbors is alive, then the cell is alive in the next state.
6. The cells beyond the Cellular Automaton's boundaries are considered to be dead.

All mentioned rules can be represented as combinations of 3 binary numbers (In the following, they are sorted by the binary value represented. 0 represents dead and 1 represents alive).

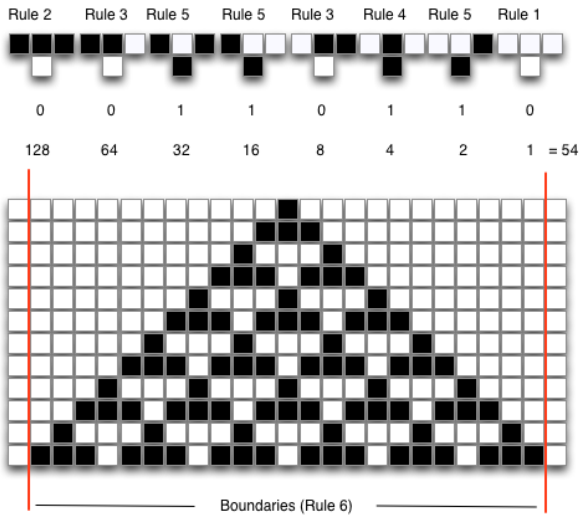

Fig. 1. Rule 54 Cellular Automaton. Each row represents one iteration at a time.

When the output shown in Figure 1 is written horizontally, it denotes a binary number, which is the number 54 in decimal representation. Hence, this automaton is named Rule 54. The rules and an exemplary evolution of a row with only one cell alive in the center of the row are shown in Figure 1.

Cellular Automata based on building rules, such as in the example above, also give the teacher the ability to scale the problem complexity in small steps. For example, the first few problems for getting started can be to ask the student to implement specific rules such as Rule 54 or Rule 30, each having their own characteristic properties -- Rule 54 is amphicheiral [33], Rule 30 is chaotic [32], Rule 110 has been shown to be capable of universal computation [30, 31]. The usual algorithms employed to program a cellular automaton, typically use iteration or recursion on discrete time units, such as the number of days of evolution undergone. More advanced participants may even deduce an elegant closed form solution for certain

problems such as Rule 54, in which the decimal value of the *n*th iteration is given in closed form by:

$$a(n) = \begin{cases} \frac{7}{15}(4^{n+1} - 1) \text{ for } n \text{ odd} \\ \frac{1}{15}(4^{n+2} - 1) \text{ for } n \text{ even} \end{cases}$$

A follow up task for the students might be to code a general elementary cellular automaton, which takes the rule number as an input. Finally, the students can be asked to add an interactive display. Thus, the teacher is provided with a plethora of settings to tweak the complexity of the problem depending on the need of the situation.

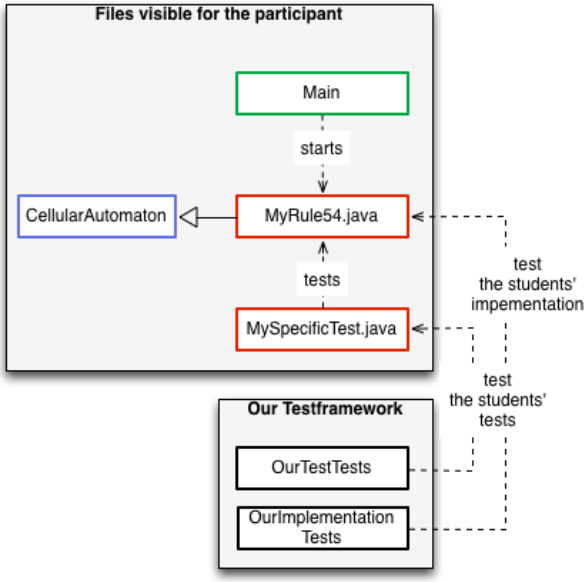

Fig. 2. Next to the part that is visible to the participants, the assignment contains a hidden part, which mainly consists of two types of tests: 1. tests that test the behaviour of the participants' implementation and 2. tests that test if the tests that have been provided by the participants test the correct things. They have to catch a certain amount of errors and have to pass against the correct implementation. The files that have been provided to the students were an abstract class that defined the required methods to be implemented (blue border), a starter class (nothing needed to be done here-green border), and the scaffolds for a test class and the according implementation class (red border). The red-bordered classes had to be completed by the students.

In our course we employed the problem of Rule 54, which allowed us to test varied classes of test cases. These classes ranged from test cases that check the

implementation on the main paths of the code to those that check the numerous edge paths, such as single celled or double celled automata. The original design of the problem included an additional feature, which allowed the Cellular Automaton to be in a continuum having the cells that are bordering the automatons limits wrap around and lie next to each other instead of having the space beyond the limits of the automaton being populated by dead cells. This feature required a larger number of unit test cases to check the additional paths of Cellular Automaton wrapping, but since the course was targeted towards a novice audience and was supposed to run only for two weeks, we decided not to include it in the final problem statement.

6.2 Testing and Assessment

In the assessment of student submitted programs, there has been a wide use of dynamic testing using a battery of unit test cases that characterize and differentiate the correct solutions from the wrong ones [26, 27, 28, 29]. Dynamic testing provides precision in measuring correctness, but is not comprehensive. To come up with an exhaustive set of test cases that catches every possible mistake that a student can make is infeasible. Hence, there is an inherent necessity for static code analysis to assess students' solutions comprehensively.

In our JUnit course, we encouraged the participants to follow the test first approach. According to this approach, tests are written first, then the actual solution is implemented. The participants iteratively improved their solutions as well as their tests, until they finally submitted their work for assessment. For the evaluation of these submissions we assessed their solution as well as their tests. To evaluate their solution, we used a battery of test cases that solely check the correctness. We are currently working on a more comprehensive assessment strategy by means of static code analysis.

Our online assessment platform is configured to allow the solution as implemented by the participants to be run for a maximum time of 20 seconds. The better implementations, thus, had plenty of time to succeed on all our assessment tests. As a side effect the timeout acts as a filtering mechanism to weed out incorrect solutions.

When the participant clicks on the "Score" button, our test suite is run against the participant's solution of the problem and then our tests are run against their tests as shown in Figure 2.

Dynamic Analysis of JUnit tests using Mutation testing

To provide thorough coverage for all possibilities, the Cellular Automata exercise would have required a large amount of test cases. To keep the workload for the teaching team in the zone of feasibility, we turned to the use of mutation testing, a common technique used in the software industry to evaluate the quality of software tests. The participants' test submissions were analyzed on the two dimensions of correctness and thoroughness. Participants' test solutions were not only required to pass a minimum number of tests against our correct "gold" solution, but it was also required that at least one or more of their test cases failed against a certain minimum number of our intentionally incorrect/mutated "coal" solutions. We required students

to pass only a minimum number of test cases against gold and catch only a minimum number of coals as opposed to 100 percent or complete correctness and thoroughness in order to relax requirements as the course was targeted towards novices. Students were informed that their tests would be tested against a correct implementation as well as multiple other incorrect implementations. Code stubs, which included an instantiated constructor of the classes of gold or coal solutions, were provided to the students with appropriate directions on how to use them.

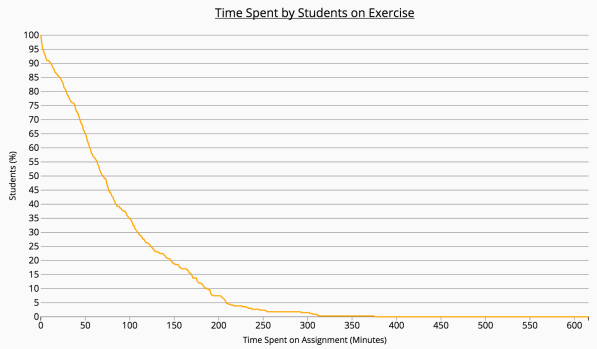

Fig. 3. The average time spent on the assignment by the participants was about one and a half hours. In comparison, we, the members of the teaching team spent a total of about 10 hours on the platform alone to create, maintain and troubleshoot the assignment. Additionally, we spent at least as much time that has not been recorded on designing and implementing the assignment.

6.3 Feedback and Reception

We have not done a formal evaluation on the participants' perception of the Cellular Automata assignment. The discussions in the forum, however, ranged from "the exercise was way too easy, it should have required us to write more test cases" to a couple of very detailed discussions that showed that the involved participants had problems with the task but were eager to solve it.

We have, however, some data on the way the users interacted with the task. 364 course participants started the Cellular Automata assignment, which was one of the main alternatives to earn the Record of Achievement. We recorded 26.968 intermediate submissions. 279 participants finally submitted the assignment. The average score for the assignment was about 90%. The time that has been spent by the participants on the assignment ranged from about 1 minute to about 10 hours, counting only those that at least solved it to some extent. One minute to solve the exercise seems to be hardly impossible with "legal" means. It was rather difficult, although not impossible to download the code and work on it offline as it contained a couple of hidden files used for testing. We, therefore, cannot eliminate the possibility that some of the very fast users have been cheating. Another explanation would be that they have teamed up to work on one of the participants' assignment and then just had to copy/paste and resubmit the solution for the other user. As we always encouraged the participants to collaborate, this would be a perfectly acceptable

approach. The number of candidates is low anyhow. No more than about 5-6 participants finished the assignment in less than 5 minutes.

7 Required Adjustments

Creating programming tasks, using cellular automata or not, requires much time and effort on the side of the teaching team. Particularly, writing sound, but not too rigid test cases is very important for the learning impact of these exercises. Sloppily designed test cases rapidly result in frustration among the participants, particularly among the novices who cannot yet distinguish if they are just not getting it right or if the tests are wrong. For the teaching team, this situation is problematic as well; soon the users' requests for help need to be handled, exercises and test cases need to be fixed, discussions in the forums need to be calmed down, etc. In the following sections, we will introduce two approaches to cover this problem from different angles. In Section 7.1 we will discuss how to increase the pool of well-maintained, high quality programming exercises by offering a platform to share these exercises among teachers.

In Section 7.2 we will discuss how replacing or enriching the current approach of dynamic testing by means of static code analysis can simplify the creation of coding exercises and improve their quality.

7.1 Exercise Management and Exercise Sharing

A coding exercise repository has been on our mind since we started to create our first programming course. During the preparation for the course on Test-driven Development the need for such a repository grew more urgent. This course raised the bar quite a bit as it's topic required us not only to test the code of the participants but also the tests that they have written. Not only did we need more tests, but also the tests' complexity grew. In discussions with colleagues, we soon realized that we were not the only ones who are facing this problem. One of the challenges for such a repository is that there are many auto-graders out there and the exercise repository needs to be flexible enough to allow exporting and importing from at least the more popular ones as otherwise it would not be widely used, thus degrading it to a mere management tool for our own purposes and not contributing to the solution of our problem.

We, therefore, analyzed two other auto-graders to find a common basis of required data to be stored with each exercise. Praktomat[4], developed at the KIT in Karlsruhe, Germany and INGInious[5], developed at the Université Catholique de Louvain in Belgium are, as well as CodeOcean[6], our own auto-grader, open source projects on github (see footnotes). Finally, we looked for a standardized data exchange format

[4] https://github.com/KITPraktomatTeam/Praktomat/

[5] https://github.com/UCL-INGI/INGInious

[6] https://github.com/openHPI/codeocean

that serves our purposes. The idea here was not to reinvent the wheel but rather to use a format that is already developed by a community of possible future users of this platform. The common formats that first come in mind, such as Common Cartridge by the IMS Global learning consortium[7] or the IEEE Learning Object Metadata (LOM) [38] didn't really meet our requirements as they are serving different purposes. We finally found the ProFormA-XML format, which perfectly suits our needs. [39]

Further developments of this Coding Repository, which is also available open source on github[8] will be covered in a future paper.

7.2 Static Code Analysis

Quality in Software Engineering refers to either quality in the functional aspects or to the structural aspects of the software. Functional aspects reflect the correctness of the software solution, which is essentially a measure of how well the software conforms to a given design, based on functional requirements or specification. Structural aspects on the other hand refer to other non-functional aspects of the software such as robustness, maintainability, that support the delivery of the functional requirements. Our testing solutions are currently exclusively through dynamic testing using a battery of unit test cases or through manual evaluation by means of peer assessment, where feasible. The emphasis has been on checking functional aspects of code solutions so far, while the structural aspects have been largely left behind. While dynamic testing does provide great precision in measuring correctness, they are not comprehensive. It is very difficult to come up with an exhaustive set of test cases that catch every possible mistake that a student can make.

Many programming languages provide tools that test the code on behalf of pre-defined quality metrics, code-style and best practices. The huge advantage of this approach is that these metrics need to be defined only once per programming language to be taught and then can be employed for all programming exercises the same way. In contrast to this, dynamic tests have to be written for each exercise separately and often have to be adjusted when the exercise is changed.

Particularly for peer assessed programming exercises, additionally a security aspect comes into play. Currently, we ask our students to run the code of their peers on their home computers to check if the running program fulfils certain requirements. This comes with a certain risk. If the grading is based on a static code analysis, the peers would not have to run the code of their peers anymore. Instead they just need to run a static code analysis on the submission. Running such a static code analysis is way less risky than running the code of more or less untrusted persons.

Moving away from functionality as the mere target of assessment, opens the door towards other important aspects of learning to write clean code:

Reliability – measures the risk of potential application failures and defects injected due to modifications made to the software. By means static code analysis, reliability

7 https://www.imsglobal.org/cc/index.html

8 https://github.com/openHPI/codeharbor

can be measured e.g. in terms of good exception handling, null pointer dereference detection, or the safe use of inheritance and polymorphism.

Efficiency – deals with time and space used by the software. It can be measured by checking for appropriate interactions with expensive and/or remote resources, data access performance and data management, memory, network and disk space management.

Security – poor coding practices and architecture increase the likelihood of potential security breaches. Issues that can be found statically are bad input validation, buffer overflows, improper locking, or SQL injection.

Maintainability – includes many subtopics, such as e.g. modularity, testability, or reusability. It not only refers to readability of code and documentation, but also to observance of design and architectural rules. Important checks with regard to ensure high maintainability are e.g. cyclomatic complexity, unstructured and duplicated code, or an excessive program size.

We've started to run our first experiments with industry strength tools for static code analysis. For now, we measured the performance requirements of such a solution employed in our auto-grader. The results so far are encouraging. These experiments will also be discussed in a future paper.

8 Conclusion

We have shown that Cellular Automata can be employed as practical programming exercises, suitable for novices as well as experts. They are a rewarding basis for developing interesting and motivating tasks. To design and to implement them, however, puts a high workload on the shoulders of teaching teams. The same applies for other interesting programming tasks as well. This is particularly true, when they are graded solely by means of dynamic testing. Alternative sources to determine a grade for the code that has been provided by a student, therefore, should be further investigated. Static code analysis is a promising candidate. Another approach is to combine automated code assessment with peer assessment. Next to simplifying the process to create programming exercises, it is necessary to provide the possibility to exchange and share programming exercises with other educators, to allow reuse and evolution of programming tasks.

9 References

1. T. Feldman and J. Zelenski, "The quest for excellence in designing cs1/cs2 assignments," in *ACM SIGCSE Bulletin*, ACM, vol. 28, 1996, pp. 319–323.
2. A. Carbone, J. Hurst, I. Mitchell, and D. Gunstone, "Principles for designing programming exercises to minimise poor learning behaviours in students," in *Proceedings of the Australasian conference on Computing education*, ACM, 2000, pp. 26–33.
3. K. Becker, "Teaching with games: The minesweeper and asteroids experience," *Journal of Computing Sciences in Colleges*, vol. 17, no. 2, pp. 23–33, 2001.

4. M. Guzdial and E. Soloway, “Teaching the nintendo generation to program,” *Communications of the ACM*, vol. 45, no. 4, pp. 17–21, 2002.
5. H. A. Lilly, “The use of cellular automata in the classroom,” in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, ACM, 1995, p. 16.
6. R. A. Bosch, “Integer programming and conway’s game of life,” *SIAM review*, vol. 41, no. 3, pp. 594–604, 1999.
7. M. Gardner, “The fantastic combinations of john conways new solitaire games,” *Mathematical Games*, 1970.
8. M. Gardner, “Cellular automata, self-reproduction, garden of eden and game life,” *Scientific American*, vol. 224, no. 2, p. 112, 1971.
9. M. Gardner, Wheels, life, and other mathematical amusements. WH Freeman New York, 1983.
10. S. G. Krantz, The proof is in the pudding: The changing nature of mathematical proof. Springer Science & Business Media, 2011.
11. S. Wolfram, *A new kind of science*. Wolfram media Cham- paign, 2002, vol. 5.
12. E. a. Berlekamp, Winning Ways for Your Mathematical Plays, Volume 2: Games in Particular. London, England: Academic Press, 1982, vol. 2.
13. P. Callahan, *Creating life: Conway’s life miscellany web page*, http://www.radicaleye.com/lifepage/, [Online; accessed 18-09-2016].
14. W. Poundstone, The recursive universe: Cosmic complexity and the limits of scientific knowledge. Courier Corporation, 2013.
15. R. Wainwright, “Lifeline-a quaterly newsletter for enthusiasts of john conway’s game of life,” *Issues*, vol. 1, 1971.
16. A. Weeden, “Parallelization: Conway’s game of life,” [Online; accessed 18-09-2016].
17. J. Mache and K. L. Karavanic, “Teaching parallelism with gpus and a game of life assignment,” *Journal of Computing Sciences in Colleges*, vol. 28, no. 1, pp. 200–202, 2012.
18. M. R. Wick, “Using the game of life to introduce freshman students to the power and elegance of design patterns,” in *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, ACM, 2004, pp. 103–105.
19. S.Beniak, *Creating life: Conway’s life*, http://gamedevelopment.tutsplus.com/tutorials/creating-life-conways-game-of-life–gamedev-558 , [Online; accessed 18-09-2016].
20. E. F. Moore, “Machine models of self-reproduction,” in *Proc. Symp. Appl. Math*, vol. 14, 1962, pp. 17–33.
21. J. Myhill, “The converse of moore’s garden-of-eden theorem,” *Proceedings of the American Mathematical Society*, vol. 14, no. 4, pp. 685–686, 1963.
22. G. A. Hedlund, “Endomorphisms and automorphisms of the shift dynamical system,” *Theory of computing systems*, vol. 3, no. 4, pp. 320–375, 1969.
23. T. Toffoli, “Computation and construction universality of reversible cellular automata,” *Journal of Computer and System Sciences*, vol. 15, no. 2, pp. 213–231, 1977.
24. P. Rendell, “This is a turing machine implemented in conway’s game of life,” *Website. www. rendell-attic. org/*-gol/tm.htm, 2005.
25. S. Wolfram, “Statistical mechanics of cellular automata,” *Reviews of modern physics*, vol. 55, no. 3, p. 601, 1983.

26. C. Douce, D. Livingstone, and J. Orwell, “Automatic test-based assessment of programming: A review,” *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, p. 4, 2005.
27. M. Wick, D. Stevenson, and P. Wagner, “Using testing and junit across the curriculum,” in *ACM SIGCSE Bulletin*, ACM, vol. 37, 2005, pp. 236–240.
28. U. v. Matt, “Kassandra: The automatic grading system,” 1998.
29. M. Joy, N. Griffiths, and R. Boyatt, “The boss online submission and assessment system,” *Journal on Educational Resources in Computing* (JERIC), vol. 5, no. 3, p. 2, 2005.
30. M. Cook, “Universality in elementary cellular automata,” *Complex systems*, vol. 15, no. 1, pp. 1–40, 2004.
31. M. Cook, “A concrete view of rule 110 computation,” *ArXiv* preprint arXiv:0906.3248, 2009.
32. E. Weisstein, *Rule 30*, http://mathworld.wolfram.com/Rule30.html, [Online; accessed 18-09-2016].
33. E. Weisstein, *Rule 54*, http://mathworld.wolfram.com/Rule54.html, [Online; accessed 18-09-2016].
34. F. Berto and J. Tagliabue, “Cellular automata,” in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Summer 2012, 2012.
35. A. Ilachinski, Cellular automata: A discrete universe. World Scientific, 2001.
36. J. P. Crutchfield, “The calculi of emergence: Computation, dynamics and induction,” *Physica D: Nonlinear Phenomena*, vol. 75, no. 1, pp. 11–54, 1994.
37. Wikipedia, *Continuous automaton*, https://en.wikipedia.org/wiki/Continuous_automaton [Online; accessed 8-September-2016]
38. Learning Technology Standards Committee of the IEEE. Draft standard for learning object metadata. Technical report, IEEE, 2002. http://129.115.100.158/txlor/docs/IEEE_LOM_1484_12_1_v1_Final_Draft.pdf
39. Sven Strickroth, Michael Striewe, Oliver Müller, Uta Priss, Sebastian Becker, Oliver Rod, Robert Garmann, Oliver J. Bott, and Niels Pinkwart. Proforma: An xml-based exchange format for programming tasks. *eleed*, 11(1), 2015. ISSN 1860-7470. URL http://nbn-resolving.de/urn:nbn:de:0009-5-41389.

10 Authors

Thomas Staubitz is a Research Associate at the Hasso Plattner Institute (HPI), where he is working on openHPI, the HPI’s MOOC platform.

Ralf Teusner is a Research Associate at the Hasso Plattner Institute in Potsdam, Germany.

Nishanth Prakash currently is a student at the Department of Computer Science at Brown University, Rhode Island, US..

Christoph Meinel is the CEO of the Hasso Plattner Institute and the Chair of the Internet Technologies and Systems group.

This article is a revised version of a paper presented at the 2016 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE2016), held 7-9 December 2016, at Dusit Thani Bangkok Hotel, Bangkok, Thailand. Article submitted 31 March 2017. Published as resubmitted by the authors 05 May 2017.