An Open Source Simulation Model of Software Development and Testing

Shmuel Ur[1], Elad Yom-Tov[1], Paul Wernick[2]

[1] IBM Haifa Research Labs, Haifa, 31905, Israel {ur, yomtov}@il.ibm.com
[2]School of Computer Science, University of Hertfordshire,
College Lane, Hatfield, Herts. AL10 9AB, UK, p.d.wernick@herts.ac.uk

Abstract. This paper describes a new discrete event simulation model built using a mathematical tool (Matlab) to investigate the simulation of the programming and the testing phases of a software development project. In order to show how the model can be used and to provide some preliminary concrete results, we give three examples of how this model can be utilized to examine the effect of adopting different strategies for coding and testing a new software system. Specifically, we provide results of simulation runs intended to simulate the effects on the coding and testing phases of different testing strategies, the adoption of pair programming in an otherwise-unchanged process, and the automation of testing. The model source code is available for downloading at http://qp.research.ibm.com/concurrency_testing, and we invite researchers and practitioners to use and modify the model.

Keywords. Simulation, Software Development, Iterative design, Algorithms, Management, Measurement, Performance, Design, Economics, Reliability, Experimentation, Theory, Verification.

1 Introduction

In many areas of software development, it is difficult to predict the effect of process changes. This is due in large measure to the impact of the scale of real-world development work. Mechanisms that work well in laboratory-sized experiments may or may not scale up to work in industrial-scale developments of large systems.

An example of a mechanism that needs to work well in large-scale development is testing. Current approaches include testing each module as it is completed by the programmers, usually by a separate quality assurance team, formalized testing during programming by the programmers, and the test-first strategy espoused most notably by Beck in eXtreme Programming [1] of writing test harnesses code first and then writing programs specifically to

pass those tests. Another approach to managing the resource applied to testing is to automate some or all of the tests, rather than having people run them. Whilst this demands a greater initial investment, subsequent runs are cheaper to perform. The question therefore arises as to when (if ever) the benefits of such an approach outweigh the costs.

One mechanism for investigating questions such as these is software process *simulation*. Here, an enactable, usually quantified, model is built of a process for software development. This model is then modified to reflect actual and/or proposed process changes, and the results compared with the initial case to determine whether the change seems to improve or degrade performance. We believe that simulation is the most effective way to investigate proposed process changes in large-scale developments; in view of the uncertainty of scaling up small-scale experiments, the only alternative is to conduct development cycles in parallel using each mechanism and compare the results, an approach which is not only costly but also risks introducing confounding factors such as users applying learning from one team to the work of the other. However, the results derived from simulation runs do not carry the same level of certainty as experiments under controlled conditions, in particular because the simulation model is inevitably a simplification of the actual process.

A considerable body of literature describing the simulation of software processes has grown up over time, including a number of journal special issues (see for example [13, 17]). This has included work on software testing and quality assurance such as that of Madachy [9].

To investigate *inter alia* the effects on a software process of different approaches to testing, we have built a new discrete event simulation model using a mathematical tool (Matlab) and used the model to investigate the effect of adopting different strategies for coding and testing new software systems. This paper describes the simulation model itself. Our work also examines the effects of different testing strategies and pair programming on the completion times of the coding and testing phases. The Matlab code of our simulation model is available at `http://qp.research.ibm.com/concurrency_testing`. We invite researchers to use and comment on the model, and to publish any improvements they make.

The work presented here shows how simulation-based studies can examine software process behavior in cases where experiments or real-world testing are either difficult or expensive to perform or produce results that cannot be easily generalized. This is especially noted in software activities relating to large systems and/or over many releases of a software product.

One characteristic of much of the published work in software process simulation is that the results of simulation exercises and a description of the model are usually presented but the model is typically not described completely, most often in respect of the omission of the underlying equations or the input data used for the runs presented. This may well be due to the size of the equations and/or data, but it does produce results that are difficult for other workers to check, and in models which researchers find difficulty in critiquing and improving. We have therefore decided to make the code of our model public and easily accessible, not only in the hope that the software testing community will make use of it in process optimization but also to allow other workers to critique it, and, we hope, to modify and improve it.

We regard the simulation model itself as the main contribution of this work. It is explicitly intended as a general-purpose simulation of the coding and testing phases of a software process

which can be modified to reflect any required process changes; in this, it is closer in spirit to that of Wernick and Hall [17] than other software process models which have generally been developed to represent a single process environment or a specific process change. We also believe that its usefulness to software engineers is enhanced by it having been written in an environment that is closer to the programming languages with which software developers will be familiar than the specific simulation environments used for other models. The specific results we have obtained so far are of interest, but further validation work is required.

2 The simulation model

2.1 Outline of the model

In this section, we explain our simulation model for the programming phase of a project. We assume the design has been completed and we are simulating iterative cycles for the construction of the program; these cycles continue until the constructed program implements the design.

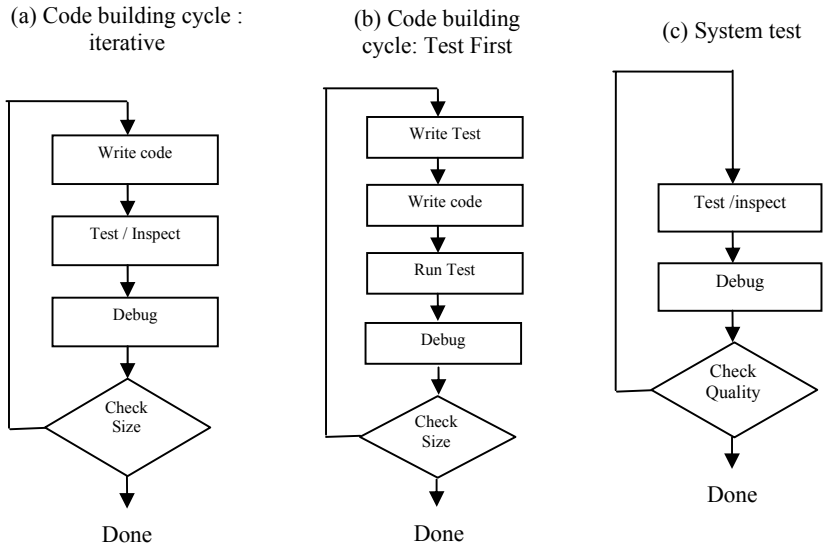

Figure 1: Simulation model structure.

We have designed the simulation model to reflect three phases of code production: code writing, testing, and debugging. First, the programmers develop the project during the code writing phase. Once this has been done, they move onto the test/inspect phase (unit, function or system after all

the code is created) where they test and/or inspect the new, and possibly the existing, code. Next, they proceed to the debug/fix phase where they debug and repair all the bugs found during the test/inspect phase. In traditional development methods this cycle repeats until the functionality of the program is complete, as shown in Fig. 1(a). In newer agile methods, the cycle repeats itself many times because each iteration is very short. Once the program is complete, the system test cycles through the test/inspect and debug/fix phases until some pre-defined quality criterion is reached, as shown in Figure 1(c). Generally, this criterion is determined pragmatically and typically reflects less than 100% freedom from bugs. The time dedicated to the code writing and testing phases is predetermined. The time dedicated to debugging depends on the number of bugs found and how long it takes to fix each one.

The Test First approach, depicted in Figure 1(b), results in a slightly different simulation. Here the tests are created first, next the code is developed, and then the tests are executed and the code is debugged. This approach is usually characterized by very short code writing cycles.

The simulation begins with the code writing phase, where objects corresponding to lines of code are actually created. These lines of code may or may not contain bugs; this is determined by a probability parameter. In the test/inspect phase, specific lines are tested/inspected and flaws may be found. In the debug/fix phase, time is spent identifying the bugs related to the flaws and some lines are replaced with new lines, which may, of course, contain new bugs. During the simulation, the program is created and improves hour by hour. In each simulated hour, one of the above activities is carried out, whether adding lines to the program, looking for bugs, or debugging and fixing the code. Each line of code is actually added as a discrete item to the simulation data so that when a specific location in the program is inspected for bugs, only the bugs that were inserted during the code writing phase are found. (We have not simulated the case of an incorrect review in which correct code is marked as a bug and changed.) In addition to explaining the above phases, this section covers the implementation of a bug to provide a more complete understanding of the simulation model.

In the real world and in our model, the more complex the program, the more difficult it is to write, test, inspect, debug, and fix. In our simulation, for the sake of simplicity, we use the size of the code measured in lines of code as a proxy for code complexity and do not take into account the type of code (scientific, GUI, etc.). Type of code would impact on the number of bugs per line, the number of lines written per hour and possibly other parameters. Sometimes code complexity is not the only issue. For example, the time passing between the introduction of the bug and its being found is a major predictor of debug time [15].

Every programming hour, the model adds ***#code_line_per_hour*** lines to the code base. This is not held as a count of lines; rather, an actual line object is created for every new program line. The number of lines of written code may be impacted by the complexity of the code (down), by the type of code (down or up), and by the programming language. For example, it is possible that GUI code is written at a much faster rate than control code. For each line created, the probability that it contains a bug is the variable ***bugs_per_line***. The duration of the code writing phase, which determines the number of lines that are written, is a parameter of the simulation run and is not part of the phase definition.

The test/inspect phase is composed of two distinct sub-phases: test writing and test execution. During test writing a number of tests are created. This number is equal to the length of the phase

divided by ***time_to_create_test***, corrected for complexity. During the test execution sub-phase, the new tests are executed in order of creation, along with as many old tests as possible. The simulation does not try to optimize the execution of specific new and old tests if there is not enough time, an important field of study in software testing [14]. However, such a module could be added to the simulation and its impact studied. Each test created has a number of parameters, some of which are used to find the lines of code actually tested by these tests. During simulation, for each test there is a percentage of new lines and of old lines covered by it. Another option that is that the test will execute a specific number of tested lines. Of the possible program lines to be tested, some are chosen at random, based on the parameters of the specific simulation run. Another parameter is the execution time per test. Manual tests tend to have shorter creation times and longer execution time, while automated tests have a longer creation time and shorter execution time.

The inspection sub-phase is very simple. The amount of code inspected is governed by the ***#lines_reviewed_per_hour*** parameter, modified to reflect code complexity. The number of lines reviewed is determined by the length of the phase.

During the debug/fix phase, any flaw found in the test phase is traced to its cause and a bug is found. The debug time is influenced primarily by the duration between the time the bug was put in and the time it was found, corrected for complexity. This is one of the better documented phenomena and is a major reason for the Test First approach [15]. If the bug was found during the review, debug is not necessary because inspection finds root causes.

In the fixing sub-phase a number of lines are modified to correct the bug. The number of lines modified may be influenced by the amount of time the bug was hiding in the code before its discovery. However, because we do not have hard evidence for this value, we have not included it in the simulation. The lines changed are in the vicinity of the bugs and are treated as new code that does not increase the program size. The time it takes to create this new code is ***hours_to_fix_bug***.

The time taken to insert, detect, and fix bugs is the heart of the simulation. Each bug is located in a specific line. For simplicity, we ignore multi-line bugs, which are more adept at evading inspection. Each bug has a probability of being discovered by a test, as indicated by ***prob_discovered_by_test***, and a probability of being discovered by inspection, as indicated by ***prob_discovered_by_inspection***. A bug has a second probability of being discovered by a test when the same test is re-executed. If it is a deterministic bug, this probability is zero or close to it (ignoring random tests). If it is a probabilistic bug (e.g. deadlock), the probability may be higher because the same input (test) might expose the bug, depending on interleaving that is usually beyond the tester's control. This means that if regression testing is undertaken in deterministic code, it rarely finds old bugs (if they become exposed to the test due to code change), and mostly finds bugs introduced by modifications or bug fixes.

2.2 Model Default Values

A common use of a simulation model is to vary one or more parameter values and observe the impact of these changes. To provide a reliable base case from which to construct the

investigations, it is first necessary to have well-supported default values for all parameters. These values are based on experimental documentation.

The values we have used for model parameters are as follows:

- ***#code_line_per_hour*** = 30 [2:207–237]
- ***Bugs_per_line*** = 0.01 [7]
- ***correction_for_time_since_placement*** = 1 + (***time_since_ bug***)/2000 – The increase in cost to fix bug due to code written between creation and fixing [3]
- ***Hours_to_fix_bug*** – base 2, multiply by 2 if a month passed, multiply by 3 if two months passed [15:6–10]
- ***Bugs per lines of code after testing*** - no default as it is a simulation decision
- ***Prob_discovered_by_inspection*** = 0.5 : Laitenberger and DeBaud [8] suggest that 70% is achievable; we pessimistically set our rate to 50%.
- ***#lines_reviewed_per_hour*** = 200 : from [8]; in our experience, these time frames differ greatly, but one hour for 200 lines is reasonable.
- ***Cost of testing*** = 1.14 hour per one hundred lines to do unit testing. [2: 146]
- ***Probability of finding bugs in test*** = .5 [15: 6–10]

2.3 Sensitivity analysis

We have conducted a sensitivity analysis to determine the effect on the base case model of modifying each of the input parameters. This analysis showed that all the parameters cause the expected model output behaviour changes when their values are modified. Our initial expectations that the completion time for the program would increase with increasing the time to fix a bug, with increasing numbers of bugs per line and with greater time required to write tests, were confirmed in simulation runs. We expected the behaviour to be different with the number of lines written per hour. If one writes very few lines per hour then the project time increases as programming takes more time. If one writes many lines, more than can be tested, many of the bugs will be discovered too late and the debugging cost will increase. We expected a 'sweet spot', an optimal value, for the number of lines written per hour, which for our simulation was found around 15 lines per hour as can be seen in Figure 2. The important factor is not the number of lines per hour but testing keeping pace with coding. If coding becomes more efficient then the testing phase has to become longer to deal with the extra amount of code generated.

3 Sample simulations

In this section, we describe three scenarios to illustrate the way in which the simulation model can be employed to examine specific issues in software processes.

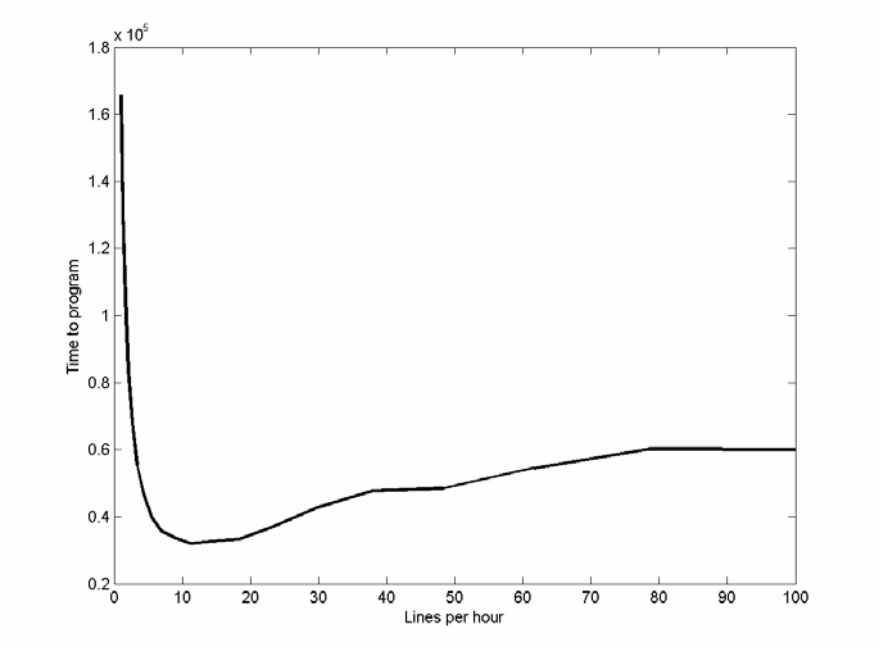

Figure 2: Total project time as a function of lines programmed per hour.

3.1 Comparing waterfall, iterative, and test first approaches

In the first simulation, we used the model to determine the optimal length of the coding phase between testing cycles. Many development paradigms are distinguished by this criterion. As our simulation runs the program is built in stages, each comprising a program/test/debug cycle, until the entire program is complete. The system test is then performed until the desired quality is reached. There are 120K lines of code and a desired final bug count of approximately 50. The total bug count is 1200 bugs for all methods, based on a probability of 1/100 that for all methods a bug is created in each line of code.

In our simulation of the traditional waterfall model, all the code is created and then it is tested. Because functions are created and tested before integration, the simulation has long programming phases of 2000 programming hours between test phases. The testing cycles are much shorter in iterative models such as the Rational Unified Process [6]. We simulate this by having programming phases of 400 hours between testing phases. In eXtreme Programming [1] using the Test First approach, tests are written as the first step and the code is tested as soon as it

is created.[1] We simulate this approach by testing after every 100 hours of programming. In our simulation, the testing cycle is always 200 hours, divided evenly between the creation of new tests and test execution, regardless of how often it is performed. As a result, in our simulation of eXtreme Programming each testing phase is longer than the programming phase to which it is attached. This division of time is not based on data from the literature but represents a percentage of testing time between 10% and 66%. Our goal is not to claim that one is better than the other, but to show that with proper management, one can optimize the length of the coding phase.

Before running the simulation, we estimated that the 2000 hour programming phase would be too long and result in a very long system test phase. We thought 100 hours (simulating eXtreme Programming) would be too short, as most of the time is spent in testing. Our results showed that with our specific simulation parameters, eXtreme Programming (simulating only the Test First aspects) works best. We believe that the main reason for this result is that the debugging time is shortest when almost no time passes between when the bug is introduced until it is found by a test. In our simulation, we see that the debugging time is indeed very small for extreme programming. This accords with our intuitive reasoning that a developer presented with a bug as they write the code would find it easier to locate and fix. Figure 3 shows the fraction of the project time spent on programming. As expected, this number decreases with time as more time is spent on testing. Also, according to accepted wisdom, the smaller the fraction of the time you initially spend on programming and the more you stress quality, the better your project will be. This can be seen when comparing eXtreme Programming with other paradigms. Less time is spent on programming initially but the progress is faster and the project finishes earlier. A counter-intuitive result, which can be seen in the long cycle (2000) line in Figure 3, is that the proportion of time spent on programming rises significantly toward the end of the project. Clearly, when the quality is lower, more time is spent on bug fixing (a programming task) toward the end of the project. Figure 4 shows the time taken to complete the project, in hours. The actual results are:

- Waterfall: 68600 hours to complete, 20800 hours to reach system test
- Iterative: 43800 hours to complete, 30000 hours to reach system test
- Test First: 27300 hours to complete, 26400 hours to reach system test

As expected, with a waterfall process, developers reach the system testing phase faster than in the iterative model; however, the system testing phase is longer and as a result the total time is longer. The unexpected (for some of us) result was that the Test First approach is so effective that not only is the system testing phase very short, but it is actually reached faster than by the iterative process.

---

[1] The implications for development timescales of the folding of design work into programming that occurs in extreme programming is not considered in this paper.

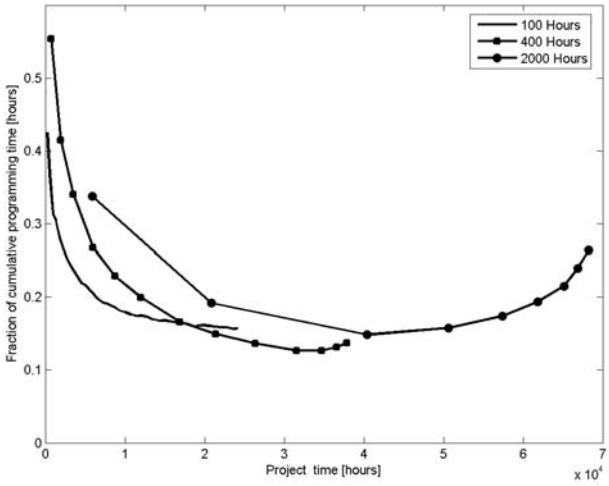

Figure 3: Fraction of the project spent on programming. Each curve denotes a different length of the programming phase.

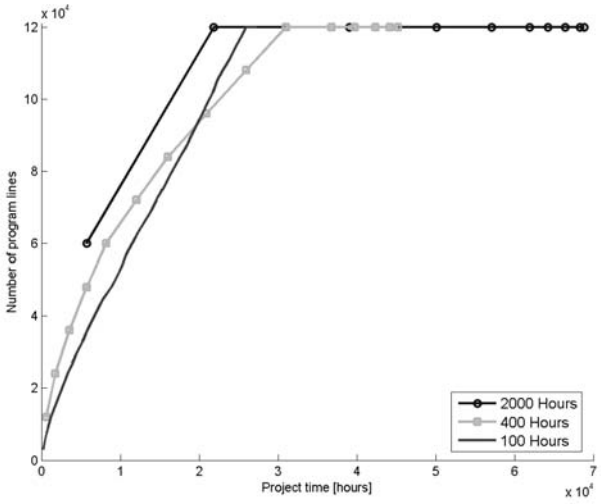

Figure 4: System size vs. programming time for three approaches to programming. Each curve denotes a different length of the programming phase.

Because this simulation runs until the bug count reaches a specified value, it is impossible to compare it with experiments where remaining bugs are counted at the end of the experimental procedure. In an experiment with programmers working under laboratory conditions, George and Williams [4] found that a Test First approach resulted in code that passed 18% more black box tests but took 16% more time. We believe that George and Williams' subjects are likely to have reached the same bug count as the waterfall users in less time. This result is reflected in our simulation, although our simulation shows a greater reduction in time than George and Williams' results might suggest.

3.2 Evaluating pair programming

Pair programming, as defined in http://en.wikipedia.org/wiki/Pair_programming, is a practice that requires two software engineers to participate in a combined development effort at one workstation. Each member performs the action the other is not currently doing. For example, while one types in unit tests the other thinks about the class that will satisfy the test . The person doing the typing is known as the driver while the person guiding is known as the navigator. It is often suggested that the two partners switch roles at least every half-hour. In this section we would like to show how our simulation model can evaluate the utility of pair programming.

We estimated that in pair programming the code generation rate would be halved, since two people are writing the same amount of code previously written by one person. Studies have been done on the amount of code produced by pairs [18], but the data relate to the productivity of the

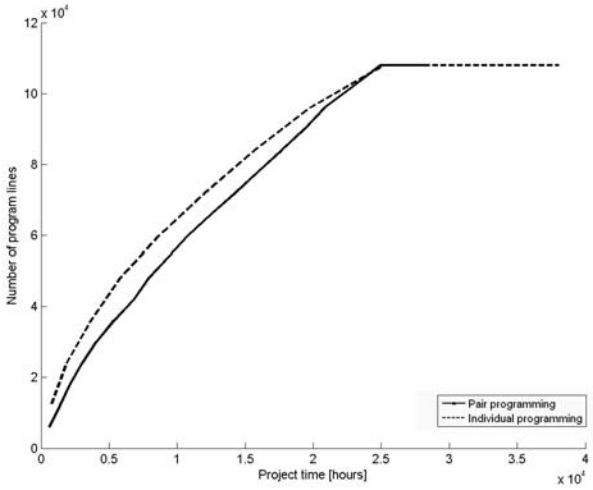

Figure 5: Program lines vs. programming type for a large project.

project, which for us is output and not input. While halving output is rather harsh, we have chosen this number as a lower bound on the basis that, if pair programming with this value is beneficial, it would be even more beneficial with a more optimistic productivity value. We also simulated a reduction in the number of bugs generated since two pairs of eyes are looking for bugs, for which we used a rate of 300/356 suggested by Williams *et al.* [18].

We estimated that in pair programming the code generation rate would be halved, since two people are writing the same amount of code previously written by one person. Studies have been done on the amount of code produced by pairs [18], but the data relate to the productivity of the project, which for us is output and not input. While halving output is rather harsh, we have chosen this number as a lower bound on the basis that, if pair programming with this value is beneficial, it would be even more beneficial with a more optimistic productivity value. We also simulated a reduction in the number of bugs generated since two pairs of eyes are looking for bugs, for which we used a rate of 300/356 suggested by Williams *et al.* [18].

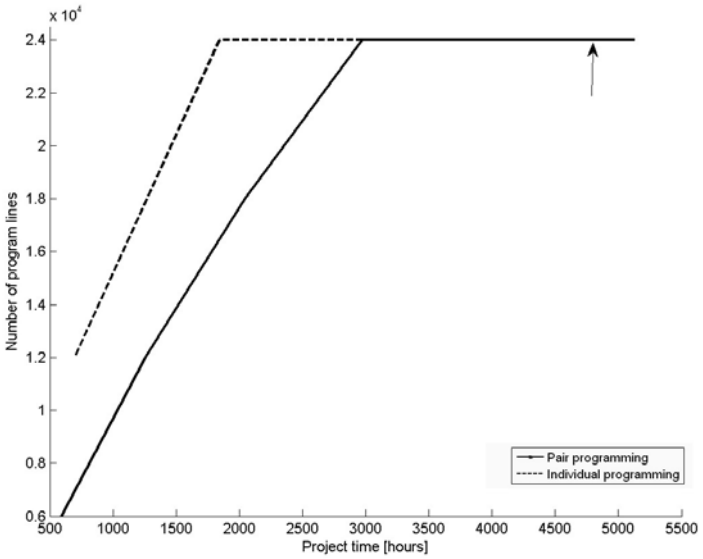

Figure 6: Program lines vs. programming type for a small project.

Our simulation showed that the gain or loss in productivity depends on the project size. In larger projects, as shown in Figure 5, careful programming is highly rewarded — not only is the total project time faster but the system test phase is reached earlier due to the decreased amount of debugging. For smaller projects, which have been studied more in the literature (e.g. [18]), there is a productivity cost for pair programming. This can be seen in Figure 6 where the arrow indicates the end of the project for single programmers. Hence, while the jury may still be out on

the question of whether pair programming improves productivity for smaller projects, our simulation shows that the advantages are quite clear for larger projects. Our findings differ from those obtained buy Williams *et al.* in small-scale experiments [18], where a gain from adopting pair programming was found even for small projects. It is possible that our simulation of an industrial process differs from the experimental protocol of Williams *et al.* which was based on students' assignments, or results from their the use of student programmers.

Our results suggest that the pros and cons of adopting pair programming for any particular project depends on a number of factors not necessarily captured in small-scale, single cycle experiments such as those of Williams *et al.* [18]. In this particular they parallel the simulation-based work of Wernick and Hall [16]. In the latter case, the effect of adopting pair programming on long-term maintainability of a software system is suggested as an element that needs to be quantified as part of a cost/benefit analysis; here, system size is another aspect to take into account.

Our method of implementing the pair programming paradigm described above can also be viewed as equivalent to early testing, as fewer bugs are introduced. In our simulation, there is a heavy penalty for late debugging, as is consistent with the literature. In conditions where such a heavy penalty is not relevant, the simulation results will be different.

3.3 Evaluating test automation

Using our simulation, we have investigated whether it is worthwhile to automate tests. We simulated automated tests as tests that cost five times as much to design, compared with typical industry figures of 3 to 10 [10] but can be executed at minimal human resource cost. We have not allowed for the maintenance cost of automated tests, which can be much higher than for manual tests. In our simulation, when a test is executed more than once, the only bugs it can find are the bugs that were introduced to the code after the previous run (due to bug fixes). Onoma *et al*. [12] observe that the main reason stated for automating testing is to ensure that newly introduced bugs are found as soon as possible after their introduction to the system code.

Our model was modified to simulate the partial automation of testing adopted in test-driven development: "With TDD, all *major* public classes of the system have a corresponding unit test class to test the public interface, that is, the contract of that class … with other classes (e.g. parameters to method, semantics of method, pre- and post-conditions to method)." ([11]; our emphasis). Automating the tests results in a number of changes that impact on the simulation results, some of them in a non-intuitive way. Creating an automated test takes longer then creating a manual one as programming effort is involved. This means that, since in the simulation the resource allocated to each testing period is fixed, there are initially fewer tests performed on the code.

One surprising result of our simulation runs is that when automated tests are used programming proceeds faster. This is due to the fact that fewer bugs are found because fewer tests are executed and a lower percentage of the time is spent on debugging. However, these bugs still need to be located and fixed before the software is released, so later, during the system tests, more time is spent fixing the bugs. Another obvious trade-off is between running many tests a

few times and running fewer tests many times. Usually, one will not choose one extreme. i.e. automating all tests, over the other, but will choose to automate a number of tests and perform

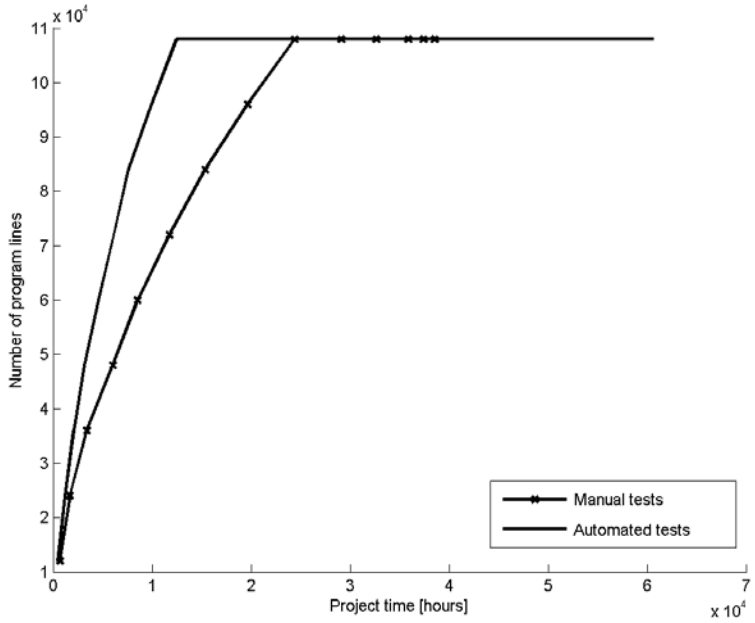

Figure 7: Performance of automated and manual testing.

the rest manually. Marick [10] states that "The cost of automating a test is best measured by the number of manual tests it prevents you from running and the bugs it will therefore cause you to miss". He also states that "A test is designed for a particular purpose: to see if some aspects of one or more features work. When an automated test that's rerun finds bugs, you should expect it to find ones that seem to have nothing to do with the test's original purpose. Much of the value of an automated test lies in how well it can do that." The cost of developing automated tests suggests that some tests should be automated and some should not. In our model we provide support for the simulation of different mixes of automated and manual tests.

In the scenario presented in Figure 7, given the parameter values we have used, i.e., an automated test is five times more expansive to write but have no execution cost, we see the benefit gained from automation is outweighed by the fact that fewer tests are initially created; while system test was reached earlier with automated testing (12,500 compared to 24,404 hours) because less unique tests were performed, the project was completed much later (60,622 compared to 38,565 hours).

Maximilien and Williams [11] have reported the results of an industrial case study using pre-written test cases for unit testing. The IBM test-driven development process examined in their report resulted in an error rate reduced by 50% and work completed on time. This was achieved with automated test cases covering 80% of the "important' classes" [11: 566]. A question that needs to be studied is whether the benefits were gained from the automation or from the investment in unit testing. Our simulation points to the latter, and poses a question regarding the use of tools like JUnit and the test automation of unit testing in eXtreme Programming. Is the practice of creating automated tests for unit testing efficient because of, or despite, the automation aspects? Maybe it is even more efficient to do these tests without the automation.

4 Summary

The goal of our work has not been to claim that Test First, pair programming and manual testing are superior to the alternatives; rather it is to show how the open-source simulation model described in this paper may be used to evaluate such claims. The research presented here demonstrates how the model can be used to evaluate software process changes, in this case testing the relative merits of different testing and programming paradigms. Using the simulation, we have obtained results which suggest that even though these approaches are justified in some situations, they may not be valid for all software development projects. For smaller programs, neither Test First nor pair programming seem always to be beneficial; test automation may be preferable when much larger programs are created. These results provide some insight when reading opinions claiming that the results of such process changes are always positive. To generalize on this observation, our simulation model can be used to predict the impact of proposed improvements on project development before these changes are tested in real projects.

Some of our simulation results can be directly attributed to the fact that the cost of finding and fixing a bug rises dramatically when a large amount of code has been written between the introduction of the bug and its discovery. If techniques such as delta debugging [5] which reduce the cost of searching for the bug become more prevalent then current simulation runs will have to be revisited.

From the experiments conducted with our simulation model, we reach a number of conclusions. First, testing early is important; in fact, the Test First approach outperforms other testing strategies. Second, pair programming may or may not improve project timescales, depending on the size of the system being developed. Under simulated conditions, larger systems perform better and smaller systems perform worse than in non-pair programming. Third, automated testing is sometimes over-rated; however, further discussion of this conclusion is beyond the scope of this paper.

5 Future work

In addition to refining our simulation model and its outputs to reconcile differences from the published results described above, we envisage that our simulation can be extended or amended to address the following:

- The implications of the need to develop test code for automated testing. In modern testing, the testing code is itself a development project. We need to model test creation as a project with its own bugs and costs. This is a fairly natural extension of the model in which two related projects are developed concurrently.
- The effect of adopting from agile methodologies techniques other than the pair programming, automated testing and Test First examples described above.
- Evaluating the effect on software costs of varying the sizes of the components and interface. This would include an examination of definitions of the 'size' of a component more sophisticated than the number of lines of code it contains, reflecting *inter alia* the complexity of the interfaces it uses (including the code behind that interface) and the type of code (e.g. control or GUI) being developed.

References

1. Beck K. *Extreme Programming Explained.* Addison Wesley Professional, 2000.
2. Capers Jones T. *Estimating Software Costs*. McGraw-Hill, 1998.
3. Capers Jones T. *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc., New York, NY, USA, 1991.
4. George B. and Williams L.A. An Initial Investigation of Test Driven Development in Industry. Proc. ACM Symposium on Applied Computing (SAC) 2003, March, 2003, Melbourne, FL, USA. ACM, 2003.
5. Holger C. and Zeller A. Locating Causes of Program Failures. Proc. 27th International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri, May 2005.
6. IBM Rational Unified Process: Best practices for software development teams, http://www-128.ibm.com/developerworks/rational/library/253.html, accessed Dec. 2005.
7. Kramer C. (2001) Black Box Software Testing, Section: 7: The Black Box Testing Organization; available at http://testingeducation.org/course_notes/kaner_cem/ac_200108_blackboxtesting/ blackboxtesting_07_blackbox_testing_group.pdf, accessed 11 December 2005
8. Laitenberger O, and DeBaud J. An Encompassing Life-Cycle Centric Survey of Software Inspection, report ISERN, 1998, pp. 98-32; Fraunhofer Institute for Experimental Software Engineering.
9. Madachy R.J. *System Dynamics Modeling of an Inspection Based Process*. Proc. International Conference on Software Engineering, 1996

10. Marick B. When should a test be automated? http://www.testing.com/writings/automate.pdf, 1998, accessed 16 December 2005.
11. Maximilien E.M. and Williams L. Assessing test-driven development at IBM; Proceedings of the 25th International Conference on Software Engineering, 2003, p.564– 569.
12. Onoma A.K., Tsai W.T., Poonawala M. and Suganuma H. Regression testing in an industrial environment, Comm. ACM, 41 (5), 1998, pp. 81-86.
13. Raffo D., Harrison W., Kellner M.I., Madachy R., Martin R., Scacchi W. and Wernick P. Guest Editors' Introduction: Special Issue on Software Process Simulation Modelling; J. Systems and Software, 46 (2/3), April 1999
14. Rothermel G. and Harrold M. J. Analyzing Regression Test Selection Techniques. IEEE Transactions on Software Engineering, V.22, no. 8, August 1996, pages 529-551.
15. RTI The Economic Impacts of Inadequate Infrastructure for Software Testing, Final report, May 2002; retrieved from http://www.nist.gov/director/prog-ofc/report02-3.pdf on 21 June 2006.
16. Wernick P. and Hall T. The Impact of Using Pair Programming on System Evolution: a Simulation-based Study; Proc. ICSM, 2004.
17. Wernick P. and Scacchi W. Guest Editors' Introduction: Special Issue on ProSim 2003. Software Process: Improvement and Practice, 9 (2), April-June 2004.
18. Williams L., Kessler R.R. Cunningham W. and Jeffries R. Strengthening the Case for Pair Programming, IEEE Software, 17, 4, July/Aug. 2000, pp. 19-25.