My year in the "Real-World"
Sabbatical Report AY 2010-2011

Zachary Kurmas
School of Computing
kurmasz@gvsu.edu

September 16, 2011

My sabbatical developing software for Atomic Object was extremely successful. I had expected to primarily learn agile software development and testing practices. It turns out that over the course of my eight months at Atomic Object, I also learned

- two new languages,
- two mobile application platforms,
- several web application technologies (including Rails and AJAX), and
- several software development tools including IntelliJ and git.

In addition, I learned a lot about the benefits and challenges of running a business.

The skills I developed while on sabbatical will help me improve both my research and my teaching. I expect that my experience with agile development practices will both (1) reduce the time it takes me to design and implement research software and (2) improve the overall quality of the software I write. My experience with many different development and testing platforms will improve my efficiency by allowing me to choose the programming language and development platform that is best suited for a given task (as opposed to just doing everything in Java or perl). I hope to use this knowledge to help colleagues choose the best language and platform for their research tasks as well.

My experience at Atomic Object will help me improve my teaching, but in less tangible ways than I had expected. I had originally anticipated learning software development and testing techniques that I would then teach. However, very few of the software development skills I learned are ideal for an undergraduate curriculum: Effectively demonstrating the usage and benefits of these techniques requires projects too large to fit in a one-semester course. Instead, the most important thing I learned is that Atomic Object and I had very different opinions about the relative importance of topics in CS 1 and CS 2. For example, I had always spent a lot of time harassing students about their comments and documentation. In reality, code at Atomic Object has very little of either. Similarly, the first part of GVSU's undergraduate curriculum focuses on object-oriented design; however, the code I wrote at

AO uses relatively few instance variables because collections of static methods are much easier to test. I also observed how being a computing professional — especially at a business like AO — requires much more than coding skills. Intellectually, I knew that non-computing skills, such as customer relations, are as important as coding skills to companies like AO; but it was still informative to actually see the different non-programming roles AO's developers take on.

1 My assignments

During my time at Atomic Object, I contributed to four projects:

- *Smart Fit*: I spent two weeks pairing with a developer writing an Android application that helps determine the optimal settings for a car seat. This application was part of a prototype presented at the Los Angeles auto show. This project was my introduction to Atomic Object's culture and development practices. This project also introduced me to the Model-View-Presenter design pattern, mocking, dependency injection, and Android development.

- *MySpectrum Android app*: I spent several weeks working on the "back end" of Spectrum Health's Android application. My role was primarily to write code to interact with Spectrum Health's SOAP server, receive the data, and package that data into Java objects to be used by the GUI. This project greatly improved my ability to effectively use test-driven-development and mocking. I also had the opportunity to make some changes to the open-source `kSOAP` library.

- *MySpectrum Blackberry app*: After the Android application was finished, I joined a team that ported the Android application to the Blackberry platform. This was challenging because the Blackberry platform is based on Java 1.3. I found a program called Retroweaver that converts Java 1.6 bytecode into Java 1.3 bytecode. However, Retroweaver does not replace libraries (e.g., Java collections and reflection); thus, I still had to modify much of the code. One major modification was removing the annotation-based Google Guice dependency injection framework. This was actually a very useful exercise because I didn't fully understand the benefits of dependency injection until I had to remove it. The Blackberry platform also does not come with the standard Java Collections library. I ended up writing interfaces in Java 1.6 that mirror the key features of the Java collections library and provided my own Java 1.3-based implementations. The process of getting Retroweaver to work with my own Java 1.3 code taught me a lot about the finer details of Java class loading.

- *LifeConnect*: I spent the majority of my time at Atomic Object working on a Rails application called LifeConnect. LifeConnect was the first project I have ever worked on that was so large I couldn't understand the entire project. Thus, it was the first project I have ever worked on where I wrote and modified code without understanding all of the potential interactions with other code. This project taught me the true benefit of many of Atomic Object's standard business and development practices.

I spent my last three weeks as the unofficial lead developer on LifeConnect. The focus of this three weeks was to make final adjustments and bug fixes before the last deployment. During this time, I got to put into practice what I learned about setting priorities, communicating with the customer, and managing customer expectations.

2 What I learned

Here's the list of the new skills and tools I learned while at Atomic Object:

- Tools
  - The Git version control system
  - The IntelliJ IDE
- Mobile Platforms
  - Android
  - Blackberry (including digitally signing applications and submitting them to App World)
- Design Techniques
  - The Model-View-Presenter design pattern
  - Dependency injection (including Google Guice and RoboGuice)
- Web Application technologies
  - Ruby on Rails
  - Javascript
    - jQuery
    - Ajax
- Testing
  - Test-driven development
  - Behavior driven development
  - RSpec (for Ruby)
  - Mocking
    - Powermock (for Java)
    - Mockito (for Java)
    - Mocha (for Ruby)
  - Cucumber (for system tests in Ruby on Rails)

- XPath (used both with JavaScript and for examining web pages when writing system tests)
- Textile (a markup language that I used to quickly generate HTML for a blog post)
- How to debug and fix open source projects
- Lots of details about how a small business runs
  - hiring
  - orientation
  - culture
  - taxation
  - profit sharing
  - employee ownership
  - accounting / analyzing profit and loss

3 Tangible classroom benefits

My experience at Atomic object will help me improve my teaching. Some of what I have learned clearly should be added to our curriculum; some topics should be briefly introduced; some topics would be beneficial elective topics; and a few topics would be difficult to include in an undergraduate course.

3.1 Topics to cover thoroughly

We must, without question, spend more time teaching our students how to test their code and then insist that they do. All code, even a quick-and-dirty program written for personal use, is completely worthless without some assurance that it works correctly. Consequently, a person's programming skills are worthless unless they also have the skills to demonstrate the correctness of the code they write. Students should learn how to articulate what the code they are about to write should do and also learn how to efficiently demonstrate that the code does, in fact, behave correctly.

Some students, after being shown the basics, can effectively teach themselves to program through trial-and-error. Most, however, cannot learn to program effectively without good professional instruction. Likewise, some students can teach themselves to become good testers through trial-and-error; however, most can't. Given the importance of testing to writing good code, we shouldn't leave that to chance any more than we leave learning how to program to chance.

There are two key aspects to teaching testing: Teaching students how to test, and teaching students what to test.

Teaching students how to test: Students need to learn how to automatically test their code. Currently students tend to test their code by "playing with it" until they either

find a bug or get bored. They often get bored long before they find all the bugs. Also, they often will introduce one bug while fixing another. Without an automated process of testing, these newly introduced bugs often go unnoticed.

Teaching students what to test: Learning to think up good test cases takes of years practice and experience. In fact, for many, it takes longer to become a good tester than it does to become a good programmer. Thus, we should get our students thinking about test cases from day one, so that their testing skills can grow together with their programming skills.

Test-driven development is the discipline of writing test cases before writing code. Programmers specifically define what must go *right* with the code before writing it, rather than trying to think up what could go *wrong* with the code afterward. Test-driven development has several advantages:

- The process of writing out the tests often lays out the structure of the code, thereby making it easier to write. (This may help those students who frequently say "I know what I want the code to do, I just don't know how to write it.")
- It forces students to think about the entire method or program before starting to write code. This forethought can help avoid the error-prone process of hacking in fixes to corner cases.

Integrating test driven development throughout the curriculum will be challenging. First, it will take time. Some of that time will be offset by student's lower cognitive load.[1] However, it will likely be necessary to abbreviate or drop a couple topics in CS 1 and CS 2. Second, it is difficult and time-consuming to automatically test event-driven and GUI-based programs. Thus, we would have to move coverage of GUI and even-driven programming to the end of CS 2 or later. There are reasonable ways to automatically test these programs, but they are not suitable for students who are still learning how to program.

3.2 Topics to introduce

Many of the topics I learned on sabbatical should be introduced to students, but not covered in depth. It is important for students to know about these tools and techniques, but they would be difficult to cover and evaluate thoroughly in an undergraduate classroom.

Mocking: Mocking is a technique for isolating one class or method for testing purposes. I believe that undergraduates should be introduced to the concept of mocking; but, that it shouldn't necessarily be taught in depth. First, it is not a fundamental computing concept; it is simply a tool to simplify testing. There are currently more valuable computing topics than can be taught during a four-year college program, and priority should be given to those topics that represent fundamental concepts. Second, undergraduate students write relatively few programs that are large enough to benefit from mocking; thus, the details of mocking may be more effectively learned on the job in the context of a large project. Also, it will be

[1] When done properly, test driven development makes the actual writing of code faster and easier, thus leaving the students with more time and energy to study.

difficult to get students to take the time to set up and use a mocking framework on a project that can be effectively tested without it.

Mobile platforms: Students tend to learn to program for a particular platform (e.g., Java Swing) with effectively unlimited resources. It is beneficial for students to get experience programming in different environments so that they truly understand that there are different programming environments; however, as with mocking, a mobile phone platform is not a fundamental concept; therefore, we should not comprehensively cover a mobile platform at the expense of fundamental topics.

3.3 Possible examples

Many of the tools and techniques I learned on sabbatical represent just one of several examples that could be used to teach a critical skill.

IntelliJ: Students should be expected to use an IDE beginning in CS 2, and should, over time, be shown how to use the IDE to help them efficiently write and debug code. The advanced features of many IDEs are very helpful, but are not easily discoverable: Students won't even know they exist unless they are shown. IntelliJ is one of many IDEs. Others include Eclipse and NetBeans.

Git: Git is a version control tool. Students should get in the habit of using version control early in their major. They should certainly begin using it as soon as they begin working on group projects. Git is just one of many tools (including CVS, and Subversion) that we could use to teach students about version control.

Rails: Rails is one of many tools used to simplify the process of (1) receiving HTTP GET or POST requests, (2) routing them to a piece of code that constructs a response (often in HTML), and (3) returning the response to the client. All students should learn the basic structure of a web application. Other frameworks include Drupal, ASP, and CakePHP.

Ruby: Ruby is one of many languages that could be included in a programming languages course. It is of particular interest because it (1) includes true closures, (2) allows the passing of code blocks as parameters, and (3) has some properties that give it a very "functional" feel.

Javascript: Javascript is interesting for many of the same reasons as Ruby. In addition, it uses a "prototype"-based inheritance system as compared to the traditional object-oriented definition-based inheritance.

Model-View-Presenter: Model-View-Presenter is a design pattern similar to Model-View-Controller. It is used to (1) separate user interface code from application logic, and (2) reduce the coupling of different application components. MVP could be taught as one technique to help students effectively test logic in business applications. It may also be a good example to use when covering design patterns.

3.4 Elective

Some of what I learned does not reflect fundamental computing principles, but would be beneficial for students to learn if they have the time and interest:

Blackberry: The BlackBerry development platform is built on top of Java 1.3; thus, many modern Java features (generics, annotations, autoboxing, collections, etc.) are missing. Also, as a mobile platform, external interactions are restricted for security reasons. Learning to program for the Blackberry doesn't teach any fundamental skills, but it gives students practice in working in a restricted environment. Learning to work around those restrictions and figuring out how to do common tasks in new ways can help develop problem solving skills.

Web Application frameworks: Although not a fundamental computer science concept, it is certainly reasonable to offer students an elective course in web programming that would cover things like JavaScript, jQuery, xpath, Ajax, rails, and other frameworks.

Dependency injection: The Google Guice dependency injection framework for Java uses reflection and annotations to automatically build aggregated objects. For example, suppose a Car object contains an Engine and four Wheels; and the Engine contains Piston objects. The Guice library eliminates the need to write a Car constructor that explicitly creates Engine and Wheel objects and an Engine constructor that explicitly creates Piston objects. This is especially helpful when refactoring. Suppose the Piston object's constructor requires a "diameter" parameter. That parameter must be passed through both the Car constructor and the Engine constructor. The bigger problem is that changes to the definition of the Piston's constructor must be then propagated through both the Engine and Car constructor. A dependency injection library handles these changes automatically. This may be an interesting topic to cover in an advanced software engineering course, or to a select group of students as part of a capstone. However, I question whether most undergraduates would be able to truly appreciate the benefits of dependency injection because (1) most undergraduate projects aren't large enough to make refactoring by hand particularly cumbersome, and (2) small class projects rarely undergo any serious refactoring, whereas the benefits of dependency injection don't become obvious until a project has been refactored several times. I didn't truly understand the benefits of dependency injection until I had to port an Android project (using RoboGuice) to the Blackberry platform (which doesn't support annotations). It wasn't until I had to propagate a constructor change through several classes by hand that I actually truly understood the value of dependency injection. Consequently, I believe this topic is best taught on the job in the context of a large project that depends on it.

4 Intangible classroom benefits

In addition to learning many topics that I could potentially teach, my experience in industry will provide many intangible benefits as well.

4.1 Misplaced priorities

l learned that my sense of relative importance of the material I cover in my programming classes was very different from that of AO.

Documentation and commenting: I had always spent a lot of time harassing students about their comments and documentation. In reality, code at Atomic Object has very little of either. There are several reasons for this limited amount of documentation and comments:

- Method documentation primarily serves as a "user's guide" for those who will use the methods. At Atomic Object, the only "users" of public methods tend to be the authors themselves; thus, comprehensive documentation is rarely necessary for the type of projects that Atomic Object takes on. (If they were to write a library to be used externally, then public methods would be heavily documented.)

- Atomic Object's use of test-driven development promotes the use of small, simple methods (because they are easier to test). The behavior of such methods is often easy to discern from the code and tests, eliminating the need for extensive documentation in most cases. (In fact, when test-driven-development is done correctly, the tests themselves serve as excellent documentation for the code.)

- Internal comments are used for code whose behavior is not obvious, or to warn others about hidden assumptions or side effects. These situations tend to occur rarely in well-designed code; hence, most files have only a few comments.

Before beginning my sabbatical I had required students to write extensive javadoc comments and internal comments believing that if the students didn't get into the habit from the beginning, they would never do it. In hindsight, although documentation and commenting are very important, I was over-emphasizing documenting and commenting to the point that it because busy work with little educational value. The javadoc documentation doesn't teach the students much — especially in CS 1 — because we often lay out what the methods are and what they should do. Thus, documentation is simply copying the project's instructions. Forcing students to write more than a few comments per assignment is often wasteful in CS 1 because the code is rarely complex enough to benefit from many comments. I now believe students will develop better commenting habits if we only require the use of comments where they truly make sense — even if it is only one or two per project.

OO-centric curriculum: I had come to believe that the only "modern" design style was Object Oriented design. I couldn't conceive of why anybody would use anything else (e.g., structured programming) unless forced to by the development environment (e.g., embedded programming). Atomic Object uses objects with instance data sparingly, and inheritance even less. It is difficult to test objects with state and methods with side effects because (1) it takes a lot of work to set up the tests and (2) complex state can lead to an exponential number of test cases. Atomic Object manages this complexity by separating data from logic as much as practical. The "data" is a class that contains mostly instance variables, getters, setters, and simple transformations (e.g., a method named "human_name" that combines a first name and last name into a single string), while the data is processed by logic objects containing class methods. Inheritance hierarchies are tedious to test because each child class has to effectively test itself and all its parents. This leads to a lot of duplication in the tests.

4.2 First-hand knowledge

Before I went on sabbatical, I would frequently emphasize the importance of certain topics by repeating stories I had heard my colleagues with industry experience tell. I can now give examples from my own experience, which should be much more credible with the students.

For example, I can tell them that I spend about 50% of my time writing tests and then justify that time with specific examples of how the presence of my tests saved me both work (by quickly identifying that a new piece of code had broken an old piece) and embarrassment (by preventing some subtle bugs from being released into production).

I also learned, much to my surprise, that Atomic Object is not perfect. Both bugs and gross code make it into production. It was especially interesting and informative to see how gross code makes it into production. Before working at Atomic Object, I had always believed that gross code only made it into production as a result of incompetent coders or unreasonable management. In fact, agile development practices can lead to small amounts of gross code when a new feature interacts awkwardly with an existing feature, but not quite awkwardly enough to justify the time and expense of refactoring. This situation is most common during times when the scope is especially volatile. For example, if two features interact awkwardly, but there is a chance a third feature will soon come into scope with similar issues, then it makes sense to wait for clarity on the third feature before refactoring. If the third feature then gets pushed down on the priority list, the gross code remains.

4.3 Now I get it

This past year was my first serious experience working in industry. Before working at Atomic Object, I "knew" a lot of things academically that I hadn't really internalized. For example, I "knew" the benefits of test driven development from reading articles and attending talks; however, I still often found it difficult to discipline myself to write tests first. It wasn't until I made a few really dumb mistakes because I took a short-cut and wrote the code first that I *really* understood the benefits of test driven development.

5 Improvements to research capacity

My time at Atomic Object taught me a lot about how to more efficiently and effectively write software. This experience should improve my research capacity by allowing me to write and test research software faster. The resulting code will also be more reliable, thus reducing the time needed to obtain consistent, reproducible results. Finally, my software should be better designed, thus making it more attractive to potential collaborators.

6 Benefit to AO

Our goal was for my sabbatical to be mutually beneficial for me and Atomic Object. Although both Atomic Object and I benefited from the sabbatical, I received a significantly larger benefit than AO. I learned a lot about developing software professionally. What I learned will help me better prepare my students for work in industry and help me write better research software more efficiently.

There was one way in which I provided AO more benefit than a typical employee: Atomic Object's blog is an important part of its marketing strategy. Employees (including visiting professors) are expected to contribute at least one article per month. About half of my posts

discussed academia. For example, one post discussed the ways in which Atomic Object reflect Carl's experience as a professor. These academia-based articles generated discussion that would not otherwise appear in the blog, thereby potentially attracting new and different readers.

One popular article discussed the challenges in teaching programming to students who have neither the opportunity to study programming in high school, nor a strong natural aptitude for programming. This article became one of the blog's most popular articles with over 30,000 hits. The article generated 58 comments on the blog itself, and another 500 on Slashdot.

7 Lessons learned

Writing excellent production-quality software is more difficult than it appears. I have known for a long time that programming ability is just one small (but important) part of writing good software; but, that fact didn't really sink in until I was able to participate in the entire software development process this past year.

Perhaps the most surprising thing I learned was just how many different software packages and other tools software developers use on a single project. I knew that professional developers used a variety of tools and packages to simplify the development process and automate tedious tasks. However, I was surprised by both the number of different tools used and just how much these tools reduced development time. (It is not practical to teach most of these tools in a typical four year program, but we should make sure our graduates are well prepared to learn these tools quickly on the job.)

8 Recommendations

Based on my experiences at Atomic Object, I recommend that we

- teach students how to test code from the beginning of CS 162,
- reduce our emphases on commenting, documentation, and objects in CS 162 and CS 163 and increase focus on structured programming and test-driven-development,
- introduce all students to mock objects and refactoring,
- strongly encourage professors to consider an industry placement for their next sabbatical (although not necessarily at Atomic Object), and
- develop a culture of getting students to physically work together in a way that they discover and share new tools.

Given just how beneficial this sabbatical was for me, I recommend that GVSU consider minor changes to make industry placements easier to arrange. There is one area in particular where GVSU could specifically support industry placements: timeline management. Atomic Object is lucky if it knows what projects will be in progress six months in advance, whereas

sabbatical preparation begins 12 to 18 months in advance. My official sabbatical proposal was due in October 2009, 10 months before I intended to begin working at Atomic Object. Shortly after I submitted my application, my Dean needed me to commit to taking a one year sabbatical so he could commit half of my salary to a replacement visiting position. Thus, I was putting up to half of my salary at risk (the risk that AO wouldn't have enough work for me when the sabbatical began.) I wonder if there is a way that GVSU could help provide a contingency plan that would allow a full-year sabbatical to revert back to a one-semester sabbatical at the last minute in the unlikely event the industry placement falls through. Perhaps the Provost's or President's office could keep an administrative job on reserve. (I imagine such a job would be helping clear some sort of administrative backlog.)

Finally, I believe we should spend more time preparing our students for the non-programming aspects of software development. However, at this point, I don't see how to do this in a classroom environment. Most of what I learned at Atomic Object, I can picture learning only in a job setting. Internships are probably the best way to provide these skills. Thus, we should make sure that each internship provides students with both technical and non-technical experience.

9 Conclusion

My time in industry has confirmed some things I already believed and given me some surprising new insights.

- I am more convinced than ever that we need to treat testing as a fundamental aspect of computing and begin teaching it the day we begin teaching programming.
- Very little of what I learned at Atomic Object belongs in the core of the curriculum. Instead, most of what I learned are tools and techniques that help us apply fundamental principles.
- I learned that I was completely wrong about the relative importance of commenting, documentation, and object oriented design.