

Running Head: DYNAMIC LANGUAGES SDLC

Dynamic Languages and Increasing the Quality of
Software and Velocity in the Software Development
Lifecycle.

Mark McFadden

MBI 630 – Systems Analysis and Design

Northern Kentucky University

When reviewing the journals, magazines, web logs, and books about software development, the use of dynamic languages within agile practices are increasing in popularity (Hendrickson, 2007). While publications on dynamic languages may get media attention, do the use of dynamic languages increase the quality of software and the velocity at which it is produced within the Software Lifecycle? The purpose of this paper is to review the available data and determine the answer to this question.

First, let us examine the current definition of a dynamic language. According to Wikipedia, “Dynamic programming language is a term used broadly in computer science to describe a class of high level programming languages that execute at runtime many common behaviors that other languages might perform during compilation, if at all.” (Dynamic programming language, 2007) While the literature is clear that there is no certain definition, there are however agreed upon characteristics that make a software language “dynamic”:

Although software experts disagree on the exact definition, a dynamic language basically enables programs that can change their code and logical structures at runtime, adding variable types, module names, classes, and functions as they are running. These languages frequently are interpreted and generally check typing at runtime. (Paulson, 2007)

Another characteristic of dynamic languages is their growth in popularity with both the literature and among developers. Next we detail why dynamic languages, particularly two languages—Ruby and Groovy—are gaining popularity.

First, dynamic languages are intuitive. Yukihiro Matsumoto, the creator of one of Ruby, states:

Throughout the development of the Ruby language, I've focused my energies on making programming faster and easier. To do so, I developed what I call the principle of least surprise. All features in Ruby, including object-oriented features, are designed to work as ordinary programmers (e.g., me) expect them to work. (Matsumoto, 2001, P. 2)

Given the complexity of many development languages, a language that is intuitive better enables programmers to more rapidly construct quality applications.

The second reason for the appeal of dynamic languages is that developers are empowered to more quickly write code. Scott Hickey, lead developer of the Eclipse IDE Plug-in for Groovy states:

Often, programmers turn to languages like Groovy for building quick utilities, rapidly writing test code, and even for creating components that make up larger Java applications because of Groovy's innate ability to remove much of the noise and complexity that accompanies typical Java-based systems. (2006)

Thirdly, the popularity of dynamic languages is due to its facilitation of the unit testing. Groovy, mentioned above is an emerging dynamic language for the java platform that better enables programmers to test their code. "The fact is Groovy's relaxed Java-like syntax, its reuse of standard Java libraries, and its

rapid build-and-run cycle makes it an ideal candidate for rapidly developing unit tests.” (Glover, 2004)

Finally, given the ubiquitous nature of the World Wide Web, an enabler of a software language’s popularity would be how well it simplifies and enhances the construction of a web application:

Of all the development tasks IT departments face, Web apps are where the use of dynamic languages particularly pays off. And Ruby, Python, and Groovy are leading the way. RoR, Ruby’s killer app, is “opinionated” in that it takes a specific view of how Web apps are designed. David Heinemeier Hansson, RoR’s principal designer, calls this approach “convention over configuration,” and it can get you up and running very quickly, as long as your design fits the chosen model. Note that little in the RoR design is language-specific. Grails — based on Groovy — offers a similar design for the JVM. (Binstock, 2007)

This student has experimented with both Ruby on Rails (RoR) and Grails and has found them intuitive and useful for rapid web development.

Now, we examine if dynamic languages help increase productivity and quality in the Software Development Lifecycle (SDLC). Upon the examination of the literature on dynamic languages, its association with Agile Methodologies is apparent. From analysis to maintenance, dynamic languages are typically utilized in an ecosystem of iterative, agile software development. Therefore, the following examination of the SDLC phases is viewed within an agile methodology framework.

Given that the phases of the SDLC can be adaptive and repeated throughout the life of a project (Hoffer, et al., 2007, p. 10), the typical agile approach to the SDLC is iterative. Scott Ambler, an agile practitioner, advocates that within an iterative approach to development you work on a small amount of requirements, and then do the respective analysis, design, coding, and testing iterating between these phases as needed. One will also iterate between working on various deliverables, working on the particular artifact at the proper time (Ambler, 2007b).

First, let us look at the Analysis Phase. According to Ambler:

Agile analysis is incremental..... This philosophy supports the incremental approach favored by common agile software processes where the work is broken done into small, achievable “chunks” of functionality. These chunks should be implementable within a short period of time, often as little as hours or days. (Ambler, 2007a)

The use of prototypes within requirements structuring is commonly viewed as a favorable practice (Hoffer, et al., 2007, p. 18). Moreover, this allows the customer to achieve a better clarity of the requirements for themselves as well as for the developer. Using a dynamic language, one can more quickly move to the prototyping of a UI or a data flow. The Rails web based framework, which is built on the dynamic language of Ruby, facilitates the use of prototypes. “You’ll find a framework that delivers working software early in the development cycle....In this way, Rails encourages customer collaboration.” (Thomas, 2005, p. 4)

Concerning the Design Phase, dynamic languages are again used within an agile system. Scott Ambler states, “Agile designs are emergent, they’re not defined up front....Although you will often do some initial architectural modeling at the very beginning of a project during ‘iteration 0’ this will be just enough to get your team going.” (Ambler, 2007b) This does not mean that there is a minimal Design Phase within the Agile Methodology. According to Martin Fowler, when discussing the Extreme Programming agile methodology, this is not the case. Moreover, within agile design processes the developer must use the concepts of refactoring and design patterns within the Design Phase:

In fact XP involves a lot of design, but does it in a different way than established software processes. XP has rejuvenated the notion of evolutionary design with practices that allow evolution to become a viable design strategy. It also provides new challenges and skills as designers need to learn how to do a simple design, how to use refactoring to keep a design clean, and how to use patterns in an evolutionary style. (Fowler, 2004)

Concerning refactoring, in his book, *Refactoring: Improving the design of existing code*, Fowler defines refactoring as the following, “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure.” (p. xvi) With regard to the Design Phase, this means that the designs will more than likely need to evolve throughout the project, with the developer improving the design as new discoveries are made. Concerning software patterns, they convey many useful

design ideas that improve the design of the application. (Kerievsky, 2005, p. 1). With that stated, dynamic languages reduce the need for patterns (Ford, 2007). Moreover, the simplicity of dynamic languages makes refactoring easier (Sommers, 2007). Therefore, dynamic languages can make an agile process easy and more productive.

The Implementation Phase is where dynamic languages provide a definite benefit. The primary help that dynamic languages provide in this stage is the rapidity of construction (Hickey, 2006). For example, see the Erase Java class in Figure 1 which reads a list of names, prints the list, filters the names based on the character length, prints the size of the filtered list, and then prints the filtered list (Weirich, 2004):

```
import java.util.*;
class Erase {
    public static void main(String[] args) {
        List names = new ArrayList();
        names.add("Ted");
        names.add("Fred");
        names.add("Jed");
        names.add("Ned");
        System.out.println(names);
        Erase e = new Erase();
        List short_names = e.filterLongerThan(names, 3);
        System.out.println(short_names.size());
        for (Iterator i = short_names.iterator(); i.hasNext(); ) {
            String s = (String) i.next();
            System.out.println(s);
        }
    }
    public List filterLongerThan (List strings, int length) {
        List result = new ArrayList();
        for (Iterator i = strings.iterator(); i.hasNext(); ) {
            String s = (String) i.next();
            if (s.length() < length+1) {
                result.add(s);
            }
        }
        return result;
    }
}
```

Figure 1. Java Class

The above Erase class produces the following in Figure 2:

```
[Ted, Fred, Jed, Ned]
3
Ted
Jed
Ned
```

Figure 2. Java Class Output

Here is Groovy code (Figure 3) that produces the same results as the Java class above:

```
names = [ "Ted", "Fred", "Jed", "Ned" ]
println names

length = 3
short_names = names.findAll { s -> s.length() < length+1 }
println short_names.size()
short_names.each { println it }
```

Figure 3. Groovy code

The Java class contains over twenty lines of code to produce the desired result.

The Groovy code produces the same result with less than ten lines of code. This shows the potential increase in both the speed and the ease of software creation with a dynamic language.

The Maintenance Phase also benefits from dynamic languages by using standard processes available within the agile ecosystem. The first process, which was examined above, is refactoring. “Without refactoring, the design of the program will decay. As people change code—changes to realize short-term goals or changes made without a full comprehension of the design of the code—the code loses its structure.” (Fowler, 1999. p. 55) See Figure 4 below for an example of the Extract Method Refactoring (Fowler, 1999. p. 110 - 116).

```

class ExtractMethodSample
  def test1(a,b)
    c = 1, d = 2, e = 3

    # Begin fragment
    if (b > 4)
      a = 1
    end
    c = a
    f = d+2
    g = b+3
    h = g
    i = 5
    c += h
    # End fragment

    puts g
    if (a < 5)
      i = 55
    end
    h = 5
    puts h, e, c, i
  end
end

```

Figure 4. Extract Method Refactoring example (Norbye, 2007).

Figure 5 below shows the extracted method.

```

class ExtractMethodSample
  def test1(a,b)
    c = 1, d = 2, e = 3

    a, c, g, i = extracted_method(a, b, d)

    puts g
    if (a < 5)
      i = 55
    end
    h = 5
    puts h, e, c, i
  end

  # TODO Comment
  def extracted_method(a, b, d)
    if (b > 4)
      a = 1
    end
    c = a
    f = d+2
    g = b+3
    h = g
    i = 5
    c += h
    return a, c, g, i
  end
end

```

Figure 5. Extract Method Refactoring example completed (Norbye, 2007).

Note that the test1 method in Figure 5 is now less cluttered, more easily read, and more maintainable. Another process that is commonly utilized is Test Driven Development:

It's increasingly common for programmers to build their code test-first: if the code doesn't do something you want, first write a test that fails because of that, and then write the next test and then the next bit of code. Continue until the code does what you want. Along the way, clean up code whenever it starts to get messy, making sure that the cleaned-up code always passes all the tests. (The technical term for such cleanup is refactoring). (Marick, 2007, p. 63)

Of note is the fact that a unit-test framework is built in both the Ruby and Groovy languages (Glover, 2004; Marick, 2007, p. 64). Therefore, unit tests can be more quickly implemented within these languages.

In summary of the phases of the SDLC, dynamic languages have great potential in increasing the speed of the development lifecycle and software quality. Moreover, there is evidence that agile methodologies do improve the software development and produce more quality software (Lindvall, 2002). However, although it appears that the literature cited above verifies that the use of dynamic languages do decrease the time in the SDLC and do increase the software of quality produced, these findings are only anecdotal. Again, there is the potential for dynamic languages to speed up the SDLC and increase software

quality, but this student was not able to find any documented research to establish this as fact.

Next, let us examine if the recent, popular dynamic languages of Ruby and Groovy are currently in the large enterprise and if so, do they provide a shorter SDLC and better software quality. Before implementing a new technology within large organizations, often technology executives and managers will first ask where it is currently being utilized. While Ruby is currently finding its way into the enterprise (Schairbaum, 2007), according to Guillaume Laforge, Groovy's Lead Project Manager, Groovy is currently utilized in many critical applications in companies such as Mutual of Omaha (with several thousands of lines of Groovy), the French Ministry of Justice, and the EADS European aerospace consortium (2007). This is promising as more enterprise Information Technology (IT) managers are attentive to the hidden costs of unfulfilled business needs and are putting aside their prejudice against dynamic languages wanting to reduce the IT backlog (Binstock, 2007). By systematically matching dynamic languages to the proper projects, IT can utilize the eloquence of dynamic languages to create clear, consistent, and reusable code and therefore recognize the output benefits without compromising the stability of the enterprise (Binstock, 2007).

While Microsoft is currently working to provide an interoperable environment for dynamic languages such as Ruby in their .Net platform (Allen, 2007), given the fact that at the time of this writing it is not production ready and that Java is still the most used language within the majority of large enterprises (see http://www.tiobe.com/tiobe_index/index.htm), this student views the Groovy

language as the best contender for large enterprise adoption. In his article entitled, *The Case for Groovy*, Scott Davis states:

This verbosity is what makes me give Java low marks on “Easy to write” scale. It just comes in at the wrong level of abstraction for many common aspects. Since we’re programmers, “Easy to write” is a tough metric to overlook. It has lead many bright developers to other programming languages such as Perl or Ruby that make “easy things easy and hard things possible.” The problem with choosing a different language is that usually as the “Easy to write” slider moves up, the “Easy to integrate” slider moves down. (p. 33)

However, since Groovy “is an agile and dynamic language for the Java Virtual Machine” (Groovy, 2007), integration within the Java platform is trivial. Oracle also considers Java’s leadership within the enterprise to be a factor in its contribution to Grails, the web framework based on Groovy, and therefore grounded in Java:

Oracle sees Grails as potentially having better chances for mainstream adoption because "at the end what you generate with Grails is a J2EE app, web container managed and deployed. You can manage it in the same way you manage any other J2EE app." (Marinescu, 2006)

Finally, while it appears that dynamic languages provide promise, as previously stated, this student did not locate any empirical data that demonstrates that Groovy or Ruby is increasing the velocity of the SDLC and software quality within large enterprises.

In conclusion, while there is much anecdotal evidence that dynamic languages themselves improve the speed of the SDLC and the quality of the software produced, this student was unable to find any empirical evidence to support this claim. Moreover, we see that dynamic languages are currently being utilized in some large enterprises. However, this student did not locate any experimental verification that the use of dynamic languages increases the speed of the SDLC or the quality of software within these enterprises. Clearly more research is needed to see if dynamic languages increase the velocity of the SDLC and software quality. This can be done first within small and medium sized organizations. Then, the findings concerning the testing and verification techniques learned in the small and medium sized enterprises can be refined and utilized to conduct research within large organizations to determine if the use of dynamic languages increases software quality and the velocity of the SDLC.

References

- Allen, Jonathan (2007, 5). *Dynamic Language Runtime Announced*. Retrieved November 14, 2007, from <http://www.infoq.com/news/2007/05/DLR>
- Ambler, Scott W (2007a, 3). *Agile Analysis*. Retrieved October 30, 2007, from Agile Modeling: <http://www.agilemodeling.com/essays/agileAnalysis.htm>
- Ambler, Scott W (2007b, 3). *Agile Design*. Retrieved October 30, 2007 from Agile Modeling. Web site: <http://www.agilemodeling.com/essays/agileAnalysis.htm>
- Binstock, Andrew (2007, 4). *Dynamic languages: More than just a quick fix*. Retrieved November 1, 2007, from InfoWorld: http://www.infoworld.com/article/07/04/16/16FEscripting_1.html
- Davis, Scott. The Case of Groovy. In Neil Ford (Ed.). (2007). *No Fluff Just Stuff Anthony Volume II*. Raleigh, NC: The Pragmatic Bookshelf.
- Dynamic programming language. (2007). In *Wikipedia , The Free Encyclopedia*. Retrieved November 1, 2007, from http://en.wikipedia.org/wiki/Dynamic_language
- Ford, Neal (2007). *Ruby Matters: "Design Patterns" in Dynamic Languages*. Retrieved November 17, 2007, from Meme Agora: <http://memeagora.blogspot.com/2007/09/ruby-matters-design-patterns-in-dynamic.html>
- Fowler, Martin (1999). *Refactoring: Improving the design of existing code*. Reading, Massachusetts: Addison Wesley.
- Fowler, Martin (2004). *Is Design Dead?* Retrieved November 1, 2007, from: <http://martinfowler.com/articles/designDead.html>
- Glover, Andrew (2004,11). *Practically Groovy: Unit test your Java code faster with Groovy*. Retrieved November 3, 2007, from IBM Web site: IBM; <http://www.ibm.com/developerworks/java/library/j-pg11094/>
- Groovy (2007, 11). *Groovy: An agile dynamic language for the Java Platform*. Retrieved November 18, 2007, from <http://groovy.codehaus.org/>

- Hendrickson, Mike, Scott (2007,5). *State of the Computer Book Market, Part Four - Programming Languages Q1 07*. Retrieved November 7, 2007, from O'Reilly Radar site:
http://radar.oreilly.com/archives/2007/05/state_of_the_co_10.html
- Hickey, Scott (2006,9). *Practically Groovy: Reduce code noise with Groovy*. Retrieved November 1, 2007, from IBM Web site: <http://www-128.ibm.com/developerworks/java/library/j-pg09196.html>
- Hoffer, Jeffery A., George, Joey F., & Valacich, Joseph S. (2007). *Modern Systems Analysis and Design*. Upper Saddle River: Prentice Hall
- Kerievsky, Joshua (2005). *Refactoring to Patterns*. Boston, MA: Addison Wesley.
- Laforge, Guillaume (2007, 11). *State of the Groovy Nation*. Retrieved November 14, 2007, from: www.grails-exchange.com/files/Guillaume-Laforge-Groovy-Keynote-Grails-eXchange-2007.pdf
- Lindvall ,Mikael., Basili, Vic., Boehm,Barry., Costa , Patricia., Dangle, Kathleen., Shull, Forrest., Tesoriero , Roseanne., Williams, Laurie., & Zelkowitz, Marvin (2002). *Empirical Findings in Agile Methods*. Retrieved November 18, 2007, from http://www.cebase.org/www/researchActivities/ebase/Lindvall_cebase_agile_universe_eworkshop.pdf
- Marick, Brian (2007). *Everyday Scripting with Ruby: For Teams, Testers, and You*. Lewisville, TX: Pragmatic Bookshelf.
- Marinescu, Floyd (2006, 5) Groovy gets a contribution from Oracle; ongoing Grails contributions discussed Retrieved November 17, 2007, from: <http://www.infoq.com/news/Oracle-helps-Groovy-and-Grails>
- Matsumoto, Yukihiro (2001). *Ruby In A Nutshell*. Cambridge, MA: O'Reilly Media, Inc.
- Norbye, Tor (2007, 11). Ruby Screenshot of the Week #23: Extract Method and More Refactorings! Retrieved November 23, 2007, from http://blogs.sun.com/tor/entry/extract_method_introduce_variable_introduce
- Paulson, Linda D. (2007). Developers Shift to Dynamic Programming Languages. *Computer*, 40(2), 12-15.
- Schairbaum, Josh (2007, 5). How I'm Using Ruby in the Enterprise 2. Retrieved November 19, 2007, from <http://www.grokblok.com/2007/5/27/how-i-m-using-ruby-in-the-enterprise>

Sommers, Frank (2007). *Refactoring Dynamic Code*. Retrieved November 17, 2007, from Artima Developer:
<http://www.artima.com/weblogs/viewpost.jsp?thread=217080>

Thomas, Dave., Hansson, David., Breedt, Leon., Clark, Mike., Davidson, James D., Gehlman, Justin., & Schwarz, Andreas (2005). *Agile Web Development with Rails: A Pragmatic Guide*. Lewisville, TX: Pragmatic Bookshelf.

Weirich, Jim (2004). *Groovin with Groovy*. Retrieved November 17, 2007, from <http://onestepback.org/articles/groovy/>