Flexible, Reusable Tools for Studying Novice Programmers

Matthew C. Jadud
Computer Science Department
Allegheny College
Meadville, PA USA
matthew.c@jadud.com

Poul Henriksen
Computing Laboratory
University of Kent
Canterbury, UK
p.henriksen@kent.ac.uk

ABSTRACT

We would like more computer science education research studies to be easily replicable. Unfortunately, the tools used for data collection are often too specialized, unstable, or just plain unavailable for use in experimental replication. Here, we present two tools to aid in the replication and extension of existing research regarding novice programmers—or to support entirely new and unrelated enquiries. The first tool is specific to the BlueJ pedagogic programming environment, and provides a starting point for replicating or extending existing studies regarding novice programmers learning Java. The second tool is a portable, stand-alone web-server with a language-agnostic interface for storing data. The distinguishing feature of this server is that it is *schema free*, meaning it can easily support a wide range of data collection projects simultaneously with no reconfiguration whatsoever.

Categories and Subject Descriptors

K.3.2 [**Computers and education**]: Computer and Information Science Education

General Terms

Measurement

Keywords

BlueJ, CS Education Research, data logging, tools

1. INTRODUCTION

"If you want to know whether a duck is crossing the street, you look twice." Harry Collins, a social scientist at the University of Cardiff, gave this glib (yet accurate) summary of the role of replication in experimental research. Experimental work in the physical sciences is built upon a tradition of replication for validation, yet many studies are never replicated, simply because there is not interest, or (worse yet), studies are published lacking enough material to accurately reproduce the work described[3].

Replicating a study requires a great deal of care. On one hand, it is necessary to understand the provenance of the original study, although this information is difficult to capture[12]: why was the study undertaken? Where was it carried out? Under what conditions? Is replication today comparable, and if so, to what degree? With complex protocols, researchers must be very diligent, possibly communicating with the original researchers to make sure methods and subsequent analysis are replicated faithfully. In studies with on-line protocols (that is, where the data is collected via some automated, programmatic means), there is the additional question of replicating the tools and infrastructure for collecting and storing research data. However, this can often be difficult—sometimes because of obsolescence (perhaps no one teaching Pascal anymore[13]), and sometimes because of the complexity inherent in the data gathering infrastructure. This paper, and the tools described herein, attempt to address this problem of a replicable infrastructure for on-line protocols.

We are interested in how novices learn to program; in particular, we are interested in replicating and extending work regarding novice compilation behavior like that described in [5] and [9]. This research explored the edit-compile cycle (Figure 1) of novice programmers using BlueJ, a pedagogic programming environment for the Java programming language. Pilot studies indicated that students spent a significant amount of time dealing with syntax errors when learning to program[4]; this implies to us that students are not spending the majority of their time on the task we set for them as instructors, but instead that they are busy fighting the compiler. This appears to be especially so for weaker students, or (using the terminology of Perkins et al.) the "stoppers" as opposed to the "movers."[8]

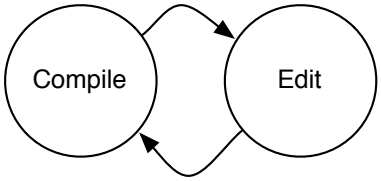

Figure 1: Novice programmers get stuck here.

BlueJ is a programming environment developed specifically for the purpose of teaching object orientation in Java[7]. BlueJ contains all the features typically found in an inte-

ICER'09, August 10–11, 2009, Berkeley, California, USA.

grated development environment: a simple text editor, the Sun Java compiler, and an integrated debugger suitable for novice use. What sets BlueJ apart from other development environments is the special emphasis that has been placed on the visualization of object-oriented programs and student interaction with those objects; through these mechanisms, BlueJ supports a distinctly objects-first approach to learning to program. Jadud's initial explorations regarding novice compilation behavior focused entirely on the changes students made in their source code from one compilation to the next. We are interested in extending this work to include student interactions with BlueJ's *class diagram*, a simplified UML representation of their object-oriented design, and the *object bench*, where instantiated objects can be interacted with and inspected by the programmer.

Unfortunately, replicating the initial studies in this area were problematic due to the complexity of the systems involved. Jadud's research required an Apache CGI server, a PostgreSQL database server, and a substantial amount of code written in Scheme (Figure 2). Reusing this framework is problematic for several reasons. Practically speaking, it is difficult to maintain all the services involved—webservers, CGI environments, and databases require care and feeding. But more problematic is that changing the type of data collected requires changes in *three* places: the client, the CGI server, and in the database itself (updating relational schema, creating new tables, etc.). These are error prone and tedious operations, not easily automated.

Ultimately, we are interested in helping BlueJ become an accessible platform for research regarding novice programmers. While students' interactions with the class diagram and object bench are interesting today, it will also be interesting to look at the debugger, unit testing framework, and other custom extensions to BlueJ in the future. Therefore, it is in our best interest (and that of the CS education research community) to support not only our "next step" study regarding BlueJ, but to develop a more flexible framework for data collection that can serve a wide variety of research needs. To this end, we have developed a stand-alone CGI and database server that supports schemaless data gathering for any application written in any language. In the next section, we describe our data harvesting server, and go on to detail one instance of a data collecting extension for BlueJ that replicates Jadud's original work and extends it to support our new inquiry regarding novice interaction with BlueJ. We close with recommendations and concerns for others interested in using these tools in their own research.

2. SERVER

The goal was to reduce the complexity of running a server capable of collecting data generated in the research regarding programmers using BlueJ. Previous data collection efforts required three pieces of software running on separate hosts—a significant maintenance cost for the researcher (Figure 2). Any change in the data gathered required changes to multiple pieces of software; in updating this infrastructure, we developed a stand-alone data gathering server that requires no configuration on the part of the researcher (Figure 3). Once started, data from any number of collection efforts can be inserted without any change to the server configuration.

The data collection infrastructure is based on a stand-alone webserver that can be executed on most any Windows, Linux, BSD, Solaris, or Mac OSX host. We currently have two implementations: the first is based on the the open-source webserver developed by the PLT Scheme research group. The PLT webserver provides performance for dynamic content (in our case, remote procedure call endpoints) that is comparable to the widely used Apache webserver[11], and is significantly easier to install and configure. A second implementation of the server has been implemented in Python, and can be executed directly on most modern Linux and Mac OSX hosts without the installation of any additional software.

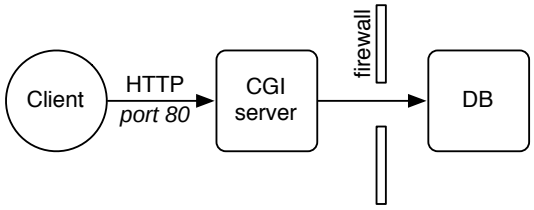

Figure 2: Original server infrastructure

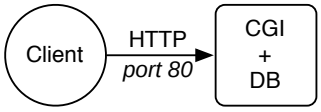

Figure 3: New server infrastructure

2.1 Supporting choice

We have attempted to design a data collection server that does not constrain the researcher with respect to their choice of programming language or data manipulation tools. Both the choice of protocol (XML-RPC) and underlying database (SQLite) are concrete examples of flexible choices for researchers interested in quickly collecting data pertaining to their research study.

2.1.1 Server API

The server API is implemented as a pair of XML-RPC remote procedure calls[15]. All reasonable programming languages have an XML-RPC library, and many languages today ship with one as a part of their standard library (eg. both Python and Ruby). A client interested in storing data first obtains a token from the server, using the `init()` remote procedure call:

init

Arguments: none.
A remote method of zero arguments; returns a plaintext authentication token.

The client then encrypts the response from the `init()` call using a secret shared between the client and server. Using this, the client can then store one piece of data into the server using the `store-secure` endpoint:

store-secure

Arguments: *string* token, *string* crypt, *array* name, *array* fields, *hash* data.
Returns the authentication token in plaintext and encrypted form using a secret key known to the client and server. The schema is transmitted as a list of column *names* and types (*fields*). The data is transmitted in a hash table of column name / value pairs.

These two methods are invoked, in sequence, for each piece of data inserted into the database.

Without authentication, a client requires only a few lines of code to store data to the server. Our complete client, with authentication and exception handling, is roughly 200 lines of Java. The server code is approximately 400 lines of Scheme, is well commented, and provides ample documentation regarding the implementation of this API. The Python server does not currently implement authentication, and is roughly 100 lines of code. As will be discussed in Section 5, our authentication provides only a modicum of security in an attempt to prevent rogue clients from inserting random data into the collection server—hence it has not (yet) been added to the Python implementation.

2.1.2 Server DB

The data collection server can use one of two relational database backends. It can be configured to either use PostgreSQL[1], a robust, open-source relational database server, or SQLite, a file-based database system[2]. SQLite is the default, and a good choice for many reasons. It is an open-source library, well documented, widely used, and it is possible to interact with an SQLite database from most any programming language. Indeed, many tools (both graphical and command-line driven) exist for browsing and interacting with SQLite database files. Perhaps most importantly, no special provision needs to be made for backing up an SQLite database—one simply copies the file, and it is "backed up." For ongoing research, having an easily accessed, shared, and archived database format simplifies data sharing and tool development over time.

2.2 Supporting change

While working with widely used and open-source tools helps future-proof a data collection effort, nothing will protect a researcher from changing data collection requirements. As a researcher goes from the initial design of a study, to a pilot, and on to a larger data collection effort, their questions may change, and therefore the data collected may change as well. This is typically a difficult proposition in traditional database designs, as the database schema must be modified, tables extended, and other changes made to the database which add complexity to the ongoing database collection effort.

For this reason, the database schema has been moved from the server (where it normally resides) to the client. A client does not just ship the data to the server, but instead it ships *both the data and the schema* to the server. This allows researchers to quickly develop data collection clients without ever reconfiguring their data collection server; they simply evolve the schema on the client until it captures the kind of data they want, and the server quietly creates the appropriate tables to accommodate each subsequent change in the schema.

As discussed in the introduction (Section 1), this flexibility was essential in developing a new data gathering client for BlueJ. Whereas the previous data collection framework required changes to the client, CGI, and database for each new piece of data, this server infrastructure allowed us to focus entirely on the data collection client. And this is where a researcher's focus should be—making sure that they are collecting the data they need to answer interesting questions, not reconfiguring CGI and database servers.

[1]http://www.postgresql.org/
[2]http://www.sqlite.org/

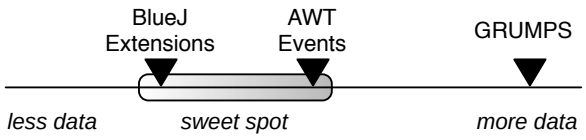

Figure 4: Finding the data collection "sweet spot."

3. A CLIENT IN BLUEJ

The goal in collecting data from an environment like BlueJ is to answer one or more research questions. While the server infrastructure presented *can* log anything, that does not mean that one *should*. Exactly how much data one needs is greatly dependent on the question being asked.

When studying the use of software written in Java, it is possible to instrument the virtual machine to provide data about *everything*. The GRUMPS project took this approach—Evans et al. developed tools for studying programmers developing software in Ada using an IDE written in Java[1]. Their work required them to develop techniques for handling millions of events in a typical logging session, from which they would attempt to distill patterns of user behavior. This approach falls into the category of "log everything, search later." We are interested, instead, in finding a "sweet spot" that involves a more targeted collection of data (Figure 4). Instead of capturing everything, we are interested in developing clients that let us focus on interactions reported by BlueJ's high-level event mechanism down through the data we can get from the Java AWT—which could yield thousands of events in a typical programming session.

The BlueJ extensions API generates two high-level classes of events:

Compilation events Generated every time a class is compiled and contains detailed information about the compilation.

Interaction events Generated whenever the user interactively invokes a method on an object or a constructor on a class. The event contains detailed information about the method that has been invoked.

3.1 What is logged?

Initial studies of novice compilation behavior began at the far left of the proposed "sweet spot;" the client presented here continues to use the high-level BlueJ Extensions API as the source for information about the behavior of programmers using BlueJ. We decided to gather both compilation

data (to replicate previous studies) and interaction data (to extend and address new questions).

Regarding compilation behavior, the BlueJ extension mechanism generates one or more events every time a student compiles a class. From a compilation event we log information such as:

- Source code of the compiled file (or files).
- Whether the compilation was successful.
- Detailed messages from the compiles (warnings and errors).
- Time spend on compilation.

To support the expanded study regarding how novices interacted with the BlueJ class diagram and object bench we added support for the logging of interaction events. These are generated every time the user invokes a method on a class (in the UML diagram) or object (on the object bench). From the invocation we log information such as:

- Method name
- ID for the object the method was invoked on
- Class the method belongs to
- Parameter types and values
- Return value
- Exceptions thrown by the method

In addition to these specific values, additional metadata to support the inquiry were captured; this does not differ greatly from that reported in [5], and includes information like the operating system, the username reported by the operating system (uniquely anonymized), IP address, host name, the BlueJ project name, a session-local sequence number for the event, and time of the event. In this particular instance, the experimental design called for the obfuscation of all data that might uniquely identify any one student.

3.2 Supporting change

As with the server, one goal for the client was to make it as flexible and as extensible as possible. There are two areas where flexibility is needed: first, in extending the client to log new types of data (eg. new events, new metadata); and second, using different protocols to store the data. Of these, the most change is anticipated in exactly what information is logged, but not how.

Changing what is logged allows for the collection of new data (and therefore the support of new studies) within the existing architecture. Figure 5 shows the relevant parts of the client architecture that support change. Researchers interested in supporting new data logging capabilities should extend the abstract class `EventData` to implement the following methods:

`String getName()` Returns the name of the data logged. This will be part of the name of the schema on the server.

`Iterator iterator()` Returns an iterator to key/value pairs of data.

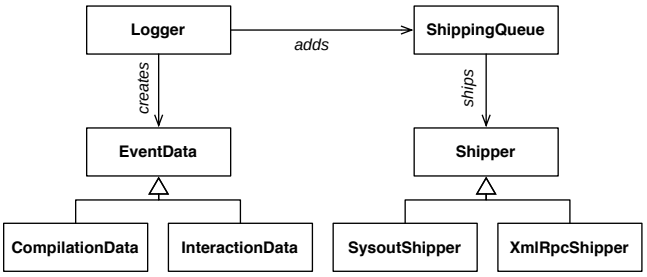

Figure 5: Architecture of the client.

`File getPackageDir()` Returns the package in which this event occurred. BlueJ has one window open per package.

`int getStartTime()`, `int getEndTime()` Return when the event started and ended.

When creating a new `EventData` object, the metadata common to all `EventData`s is automatically added to it. The key/value pairs of the `iterator()` must be of type `String` for the keys, and the values can be of the type `Integer`, `Double`, `Boolean`, or `String` (a limitation of the XML-RPC protocol). Once a new `EventData` object is created, it is placed in the `ShippingQueue`, and then shipped, stored, or ignored, depending on how the `Shipper` is implemented. The provided `SysoutShipper` prints the contents of `EventData` objects to standard out, while the `XmlRpcShipper` sends data over HTTP using the XML-RPC protocol described in Section 2.1.1.

3.3 Installing the logging extension

As a BlueJ extension, the client can be installed in several different ways. A BlueJ extension is a JAR file that is automatically loaded and started by BlueJ if placed in the correct place. Extensions can be installed on per project, per user, or per computer basis, or network-wide. Installing the logging extension per project makes it possible to only log data from one particular project and avoids logging data from other projects that a student might work on. A per user installation can be useful in a lab setting where the researcher wants to log everything a particular student is doing in BlueJ, but not other users using that computer. A per computer installation logs everything from that computer. In the case where BlueJ is installed on a network, an entire campus of users can be monitored.

The configuration file *delta.properties* must be edited, and four properties set, before deploying the client:

location For multi-institutional studies this will typical be the name of the institution. This will be part of the schema name.

server.type Fully qualified class name of the shipper class to use for storing the data.

server.address The address for the server. The interpretation of this property is done by the specific shipper chosen.

debug A boolean indicating whether debug output should be enabled.

This properties file and the associated JAR must then be dropped into either the project-local, user-local, computer-local, or network-wide BlueJ extensions folder (network-wide means that BlueJ is installed on a networked drive). The next time a project containing that extension is launched (or any project in the case of user/computer/network installations), the data logging client begins its work, shipping the results of compilation and interaction events to the described server.

4. STUDIES ENABLED

From 2006 through 2009, our schema-free server has been used to support the replication and extension of previous studies of novice programmers carried out by Jadud[5]. Because it is lightweight and can be deployed as a user on any typical Linux host, it has successfully been used for in-place data gathering at multiple research sites, eliminating the need for a DBA or similar support in setting up and maintaining a data collection server.

Tabanao et al. in 2008 successful demonstrated the replication of prior studies of novice programmer behavior using this collection framework[14]. This work is particularly interesting because it demonstrates a connection between the behavior of novice programmers (as captured by their interactions with the compiler) and their performance in their introductory programming course (as measured by coursework and examinations).

Rodrigo et. al expanded this line of inquiry to include questions regarding student affect. In particular, they have begun investigating the intersection of automatically collected data and student affect in the context of introductory programming[10]. Following on from this initial work are results indicating that data collected with our framework can be correlated to student affect. Put simply, it is possible to correlate compilation behavior with affective states like frustration[9]. This remains an open and ongoing line of inquiry.

4.1 Related and Future Work

Recently, a clean-room implementation of the data collection framework described here was published at SIGCSE 2009[2]. The results of Fenwick et al.'s replication support the findings of Jadud and Tabanao et al. Along with our successful three-year use of these tools, we take it as external validation that the kind of data we are collecting has utility in the study of novice programmers.

In continuing to expand and improve our analytical tools and techniques, we wish to support additional researchers in similar or related inquiries. This kind of data collection results in rich, complex data that can benefit from many eyes and many kinds of analysis. To this end, we have begun to develop a relationship with the Pittsburgh Science of Learning Center, and are hoping to find ways to tie the output of our data collection tools into DataShop, an open data repository developed to support the longitudinal study of fine-grained educational data[6].

5. POTENTIAL PITFALLS

Both the client and server provided a very flexible environment for replicating and extending previous work regarding novice programming behavior in BlueJ. In particular, it was invaluable being able to extend the client, test it in BlueJ, and rapidly cycle back to add new data to the logging extension (or remove data that turned out to be less-than-useful). That said, the approach to data collection described here is not without its problems.

First, the encoding of text is a constant problem. The BlueJ text editor is a Unicode-aware editor; therefore, every tool in our chain must be Unicode-aware, or we risk the possibility of data loss. Although not strictly necessary, the client takes the precaution of Base64 encoding the Unicode strings before sending them to the server along with the actual encoding used; this gives some confidence that server- or client-side localization settings will not somehow corrupt or otherwise transform the data.

Second, the server is currently only "secure" to a degree. In our current implementation, both the client and the server must be configured with a common pass phrase. Although the client is distributed in binary form (perhaps in a BlueJ project), simply running the UNIX command `strings` over this file will yield the pass phrase. Using this pass phrase, it is possible for someone to build a client capable of sending data to (but not retrieving data from) a collection server. This makes for an easy disk-filling attack, where rogue clients might store many large records to the server. Any number of solutions to this problem could be implemented, but we believe the potential value of these attacks makes them unlikely in the first instance. More robust means of securing the server are obviously possible, and will be considered/implemented as necessary.

Lastly, the XML-RPC protocol is both a liberating and limiting choice; it raises the level of abstraction at which a client can communicate with the server, but is limiting as to the types of data that can be shipped within the protocol. However, the choice of this verbose protocol allows for the easy interaction of clients and servers written in any number of languages. Short of using an ad-hoc RESTful protocol[3], XML-RPC appeared to be the most widely implemented solution available.

6. CONCLUSION

An infrastructure for flexibly and extensibly storing data is only one part of a much larger puzzle when it comes to computer science education research. That said, we have taken steps to use tools that simplify at least part of the challenge of collecting data using on-line protocols. Our choice of protocol (XML-RPC) allows for many languages and environments to serve as the client in a data gathering exercise. Likewise, our server uses a common database format (SQLite), making it accessible to a wide variety of data analysis tools.

Multiple studies resulting in Masters theses and other publications have been supported through the use of this light, flexible database infrastructure. We hope these tools will continue to provide a foundation for researchers interested in developing sharable and replicable studies regarding students and experts alike in the computing world.

[3]http://en.wikipedia.org/wiki/REST

7. OBTAINING THE SOFTWARE

All of the software discussed here (as well as additional tools developed as part of our efforts or contributed by colleagues) is open-source and is available via Subversion. Browsing to the URL `http://svn.jadud.com/ncb/` provides a web-based view of a publicly-readable Subversion repository. The README file provides instructions for downloading and using this material, describing its installation and use.

8. ACKNOWLEDGMENTS

Many thanks are due to Ian Utting, Michael Kölling, Sally Fincher, and David Barnes for their input in the development of this infrastructure. Additional thanks to Michael Hughes at Olin College, Ma. Mercedes Rodrigo, Ma. Beatriz Espejo-Lahoz, and Emily Tabanao at the Ateneo de Manila University, and Ryan Baker at Carnegie Mellon University for their support and collaboration.

9. REFERENCES

[1] H. Evans, M. Atkinson, M. Brown, J. Cargill, M. Crease, S. Draper, P. Gray, and R. Thomas. The pervasiveness of evolution in grumps software. *Softw. Pract. Exper.*, 33(2):99–120, 2003.

[2] J. B. Fenwick, Jr., C. Norris, F. E. Barry, J. Rountree, C. J. Spicer, and S. D. Cheek. Another look at the behaviors of novice programmers. In *SIGCSE '09: Proceedings of the 40th ACM technical symposium on Computer science education*, pages 296–300, New York, NY, USA, 2009. ACM.

[3] J. Giles. The trouble with replication. *Nature*, 442(7101):344–347, July 2006.

[4] M. C. Jadud. A first look at novice compilation behavior. *Computer Science Education*, 15(1):25–40, 2005.

[5] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *ICER '06: Proceedings of the 2006 international workshop on Computing education research*, pages 73–84, New York, NY, USA, 2006. ACM Press.

[6] K. R. Koedinger, K. Cunningham, A. Skogsholm, and B. Leber. An open repository and analysis tools for fine-grained, longitudinal learner data. In *Educational Data Mining 2008: 1st International Conference on Educational Data Mining, Proceedings*, pages 157–166, 2008.

[7] M. Kolling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4):249–268, December 2003.

[8] D. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons. Conditions of learning in novice programmers. *Studying the Novice Programmer*, 1989.

[9] M. M. T. Rodrigo, R. S. Baker, M. C. Jadud, A. C. M. Amarra, T. Dy, M. B. V. Espejo-Lahoz, S. A. L. Lim, S. A. M. S. Pascua, J. O. Sugay, and E. S. Tabanao. Affective and behavioral predictors of novice programmer achievement. In *ITiCSE '09: Proceedings of the 14th annual conference on Innovation and technology in computer science education*, New York, NY, USA, 2009. ACM.

[10] M. M. T. Rodrigo, R. S. j. Baker, J. O. Sugay, and E. Tabanao. Monitoring novice programmer affect and behaviors to identify learning bottlenecks. In *Philippine Computing Society Congress 2009: Research-in-Progress*, March 2009.

[11] Shriram Krishnamurthi and Peter Walton Hopkins and Jay McCarthy and Paul T. Graunke and Greg Pettyjohn and Matthias Felleisen. Implementation and Use of the PLT Scheme Web Server. *Higher-Order and Symbolic Computation*, 2007.

[12] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.

[13] J. C. Spohrer and E. Soloway. Analyzing the high frequency bugs in novice programs. In *Empirical Studies of Programmers*, pages 230–251. ACM, 1986.

[14] E. Tabanao, M. M. T. Rodrigo, and M. C. Jadud. Identifying at-risk novice programmers through the analysis of online protocols. In *Philippine Computing Society Congress 2008*, 2008.

[15] D. Winer. XML-RPC Specification, January 2007. http://www.xmlrpc.com/spec/.