Blackbox: A Large Scale Repository of Novice Programmers' Activity

Neil C. C. Brown, Michael Kölling, Davin McCall and Ian Utting
School of Computing
University of Kent
Canterbury, Kent, UK
{nccb,mik,dm391,iau}@kent.ac.uk

ABSTRACT

Automatically observing and recording the programming behaviour of novices is an established computing education research technique. However, prior studies have been conducted at a single institution on a small or medium scale, without the possibility of data re-use. Now, the widespread availability of always-on Internet access allows for data collection at a much larger, global scale. In this paper we report on the Blackbox project, begun in June 2013. Blackbox is a perpetual data collection project that collects data from worldwide users of the BlueJ IDE – a programming environment designed for novice programmers. Over one hundred thousand users have already opted-in to Blackbox. The collected data is anonymous and is available to other researchers for use in their own studies, thus benefitting the larger research community. In this paper, we describe the data available via Blackbox, show some examples of analyses that can be performed using the collected data, and discuss some of the analysis challenges that lie ahead.

Categories and Subject Descriptors

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

General Terms

Experimentation

Keywords

Blackbox, BlueJ, data collection, programming education

1. INTRODUCTION

Learning to program is a central challenge of computing education. As programming is performed on a computer, students' programming behaviour can be automatically monitored – an idea decades old [8]. There have been many prior small and medium scale studies monitoring programming behaviour at single institutions (see section 3), but the widespread availability of Internet access means that large-scale cross-institutional studies can be easily run.

The authors develop and maintain BlueJ, a Java IDE designed for beginners. BlueJ has over 1.5 million users each year, and is used widely in the first year course at universities in many different countries. This paper describes the Blackbox data collection project, which draws participants from the worldwide BlueJ user pool and records their interactions with the IDE and the source code that they write. This project provides gigabytes of collected source code, which we are making available to other researchers to benefit the computing education research community. This is not a single completed *study*; rather, it is a perpetual *data collection project* that will continue collecting data and serve as a basis for a potentially wide variety of future studies.

This paper is split into two halves. In the first half, we look at the background of BlueJ and data collection projects (sections 2 and 3), then examine the research issues of the Blackbox project (section 4) and present design decisions and initial outcomes (section 5). In the second half, we show some small examples of the analyses that can be performed with the Blackbox data (section 6), before offering discussion on the challenges in analysing this kind of data (section 7). Our contributions include:

- A discussion of research and analysis issues surrounding projects of this type (sections 4 and 7).
- Initial outcome statistics for the project (section 5).
- Several small analyses, including a replication of previous results (section 6).
- Information for other research groups on how to gain access to this data set for their own research projects.

2. BLUEJ

BlueJ is a programming environment designed to be used by those learning object-oriented programming in Java. It was first released in 1999, and now has a large worldwide user base. In 2012, BlueJ was invoked at least 15 million times, by a total of over 1.8 million users; whenever BlueJ is loaded, it attempts to make a brief one-off connection to a central server (which we term a "phone-home") to report the BlueJ version, Java version, and OS version. This connection will not always be successful; users may temporarily or permanently lack Internet access or have firewalls (on their

SIGCSE'14, March 5–8, 2014, Atlanta, GA, USA.

machine, local network, or whole country) that prevent the connection. Therefore this number only gives a lower bound on the total amount of BlueJ use – but, pertinent to this paper, it gives an accurate reflection of the number of BlueJ users where BlueJ can connect to a remote server.

3. PRIOR WORK

There have been many studies, using many approaches, of students' behaviour in introductory programming classes. Here we will restrict ourselves to discussing those that focused on their interactions with their programming environment, and the resulting program source code.

Thomas et al [9] recorded 4.7 million actions over a six-week period in 2003 from 141 students using an Ada IDE, but these were very low-level events (such as mouse clicks and key presses, captured from the GUI framework) and data-cleaning proved to be a significant problem, although they did also establish the viability of using such data to answer questions formulated after the data was gathered.

In 2005, Ahmadzadeh et al [2] collected much coarser-grained data (including source code) from 192 students in the School of CS & IT at the University of Nottingham, mostly focused on compilation errors encountered by students whilst using the JCreator environment.

Edwards et al [4] collected result-focused data from 1101 students over a five-year period. The data gathered was students' work-in-progress as they submitted their evolving programs for testing by, and feedback from, the Web-CAT tool. They eventually captured 89,879 submissions from two courses using Java and one using C++.

Jadud [6] and Fenwick et al [5] have recently used bespoke extensions to the BlueJ IDE to capture student interactions at an intermediate granularity, capturing Java compiler input and output at every invocation. Jadud captured 42,000 events from 186 students over two years, and Fenwick et al captured 55,000 from 110 students in a single year.

The questions addressed by these studies have ranged from compilation and editing behaviour to time-on-task. In all cases except Thomas et al, the particular research questions to be addressed were part of the design of the data-gathering apparatus, but in many cases the data proved amenable for use in answering other questions which became apparent only after the data had been gathered. Some of the studies have recorded very large numbers of events, some have involved reasonably large numbers of students, but all published cases involved students at a single institution. The Blackbox data differs from these prior studies in several ways, but two are particularly significant: the amount of students involved (and data collected) is larger by several orders of magnitude, and the data is shared with other researchers.

There has also been much work in the analysis of large bodies of program code, e.g. the Mining Software Repositories conference [1], but it is primarily focused on professional or open source repositories at each commit, rather than learners' behaviour at each edit and compilation.

4. RESEARCH ISSUES

The technical challenges of the Blackbox project are fairly straightforward and well understood. The more difficult issues relate to the ethical and research design side of the project; in this section we will explain the pertinent issues.

4.1 Identifiable Data

We wanted to ensure that participants would suffer no negative consequences from participating in the data collection. We could have asked participants demographic information (age, level of experience, etc)[1], but collecting this data places a responsibility on us to safeguard the data, allow individual withdrawal from the study and so on. Managing this for hundreds of thousands of users seemed untenable. Thus we made the decision to keep the data anonymous, as described in the next section.

It is possible for individual researchers to configure Blackbox to identify their own students for the purpose of a study, or to extend the data collected to include identifiable or demographic information – this is discussed in section 4.6.

4.2 Anonymisation

The two main places where identifiable information might be found in the collected data are the local path to the project and the source code.

Paths to BlueJ projects often feature an identifying username, either locally (`C:\Users\john\Documents\proj1`), or on a university network (`\\store\js123\proj2`). To mitigate this, the client transmits only the last directory in the path (above, `proj1` and `proj2`) and a hash of the full path. This allows the same project to be identified across sessions, but stops the identifying paths from being sent to the server.

Anonymising source code is a difficult challenge. Identifying information can appear in comments, string literals, and even names of variables and methods. Strong anonymisation could be achieved by blanking all comments, strings, and by renaming all variables and methods (e.g. by hashing). However, this loses much useful information; the names of methods (e.g. getHeight) can convey information, comments can be useful for analysis and so on. Therefore, Blackbox removes only the comment that occurs before a class header. This is where author names are usually entered, so this should reasonably anonymise the code without losing other useful context. We explain in the information to participants that this level of anonymisation will take place.

4.3 Data Design

One difficulty with the Blackbox project was deciding exactly which data to collect. While many studies collect data with a single analysis in mind, the Blackbox project is intended to support many different researchers for different purposes (much like a census). In an attempt to ensure that the data collected by the Blackbox project was useable by other researchers, we sought to engage interested parties at the SIGCSE 2012 [7] and ICER 2012 [11] conferences to ask for feedback on our design.

The initial design was primarily guided by the features of the BlueJ IDE, and this was then refined based on feedback received during these sessions (e.g. we now collect more fine-grained editing data than initially envisaged). We were keen to pick the right level of granularity for data collection. We believed that collecting data at a very low level (e.g. click-streams, with data on mouse movements and exact keypresses) would be too voluminous and too difficult to analyse. Instead, we work at a higher level, recording line-edits to source code, and IDE actions rather than the mouse/key movements that triggered them.

[1]Although there would have been little motivation for participants to supply this information accurately.

4.4 Data collected

The data collected for the Blackbox project includes the following:

- A persistent unique identifier, which is assigned on opting in and persists across sessions, allowing some longitudinal analyses.
- Start and end times of programming sessions.
- Use of all IDE tools, such as editing, compiling, execution, instantiation of objects, interactive method invocations, runs of unit tests, use of the debugger, use of source repositories, use of the codepad, etc. In all cases, event records include time stamps and relevant details (e.g. method invocations include parameters and return values, compilation events include compilation outcomes/errors, unit test runs include test results).
- Editing behaviour. This is collected at source line level: every time a user edits a line of code and the cursor then leaves that line, an edit event is recorded. Multiple consecutive character edits within the same line are thus collapsed into a single event.
- An optional experiment and participant identifier – see section 4.6.

4.5 Data Caveats

Although the Blackbox data is large scale, there are various notable restrictions in its utility. One obvious restriction is that the data specifically concerns using Java within BlueJ. BlueJ is often taught in an objects-first way, eschewing Java's `main` method. Thus it is a biased subset of the ways in which Java is taught. Another obvious restriction is that we do not know what the user was intending to do (e.g. their current task) or their thought processes at the time; we can only observe the resulting changes to the source code.

The data in Blackbox comes with a persistent unique identifier (a randomly assigned ID for each participant) but no further identifying information. It is possible to track a user longitudinally, but nothing will be known about that user. They may have years of Java experience and be using BlueJ for prototyping, they may be an instructor preparing materials for a lecture, or they may be a beginner who has never programmed before. We believe that the latter is the most prevalent case, but there is no way to confirm this for an individual user from the data. Instead we must rely on heuristics; for example, it may be reasonable to assume that a user who is seen first in September and never again after the following December/January was taking a single-semester Java course. In contrast, a user seen over the course of several years is likely to be an instructor.

The user tracking in BlueJ uses an identifier stored in the client's BlueJ profile. If two people use BlueJ on the same machine with the same profile, they will appear as one user. We believe this case is relatively rare, unlike the reverse: if one person uses BlueJ on multiple machines with different profiles (e.g. a home machine, a laptop and a university machine), they will appear as multiple users. There is no simple way to track this. Instead, they must be viewed as multiple users with gaps in their history. Most users are likely to have some gaps in their data recording history, due to switching machines, or their Internet connection breaking during a BlueJ session and so on. Most gaps can be identified by comparing the state of a project's source code between the end of one session and beginning of the next.

4.6 Local studies and demographics

Blackbox supports local and extended studies in two different ways. The first use case is a researcher who wishes to perform a study only on their home students. For this purpose, an experiment identifier can be generated and added to the configuration information in BlueJ. This identifier can be added by an administrator for a complete installation (e.g. all machines at an institution), or individually by a BlueJ user. Using this identifier, researchers can then identify users from only their own experiment. Experiment identifiers cannot be extracted from the database and are effectively non-guessable, so researchers cannot identify students from other institutions.

The second use case is a researcher who wishes to collect demographic or personal data about their subjects, for example to relate programming behaviour to prior experience, age, gender, etc. For this purpose, a participant identifier can be set in the Blackbox configuration. A researcher could generate participant identifiers and ask students to enter these in their BlueJ preferences. The researcher could then use the same identifiers to collect additional data relating to demographic or personal information.

It is worth emphasising that in this case the researcher is responsible for obtaining ethics approval for that study, obtaining participant consent and safeguarding the data. This extra data is not send to Blackbox, and no personal information is stored on the Blackbox servers at any time.

5. INITIAL OUTCOMES

The Blackbox project went live on 11 June 2013. BlueJ version 3.1.0 was released at that point, and the first time each user loaded it, they were presented with a dialog asking them to opt-in to the Blackbox project. Whether the user opted-in or not, the dialog is never shown again unless they explicitly invoke it via the program options.

5.1 Opt-in Rate and User Numbers

As mentioned previously, when BlueJ loads it "phones home" to a central server. By dividing the total sessions recorded in Blackbox by the number of phone-homes by BlueJ 3.1.0, we can form an opt-in rate for a given time period. After November 2013, the average daily opt-in rate is 42% (min: 37%, max: 48%). Nearly six months after launch we have a total of over 150,000 users participating in the Blackbox project, with over 10,000,000 compilation events. Based on current total BlueJ user numbers, version adoption rates and opt-in rate, we expect that during 2014, the number of users included in the data collection will be over 500,000, with over 100 million compilation events.

5.2 Infrastructure

The Blackbox server and database are supported by two machines running Ubuntu Linux. Each machine has two hyper-threaded 6-core 2.5Ghz Xeon processors with 32GB of RAM and four 2TB drives set up in RAID 5 configuration. The first machine, named *black*, holds a write-only version of the Blackbox database and collects all incoming data. This database is live-mirrored to a second machine, named *white*, where the database copy is read-only. Researchers only have access to the *white* server, ensuring that analysis cannot corrupt the original database. We are very grateful for support by SIGCSE in the form of a SIGCSE Small Grant, which supported part of the cost of these servers.

Error	% of all errors
Unknown variable	16.7
Bracket expected	10.3
Unknown method	10.1
Semicolon expected	10.0
Illegal start of expression	5.0
Unknown class	4.6
Incompatible types	4.0
Method application error	3.5
Private access violation	3.5
Missing return	3.3

Table 1: Top ten most frequent compiler errors from Jadud [6].

Error	% of all errors
Unknown variable	17.7
Semicolon expected	9.5
Unknown method	7.6
Bracket expected	6.5
Unknown class	5.3
Incompatible types	4.5
Illegal start of expression	4.4
Method application error	3.7
Identifier expected	3.6
Not a statement	3.0

Table 2: Top ten most frequent compiler errors from Blackbox, to 2013-12-01 (excl).

6. EXAMPLE ANALYSES

In this section, we will give some small examples of the kinds of analysis that are possible with the Blackbox data. All the results presented in this section are from the Blackbox data set up to 2013-12-01 (exclusive). The source code for these analyses is openly available at: `http://www.cs.kent.ac.uk/~nccb/blackbox/`.

6.1 Error Count Replication

One example of similar prior work is that of Jadud [6]. In his thesis, Jadud gives a table of error frequencies; the top ten are repeated here in table 1

We replicated this analysis, to compare the frequencies in our own larger data set. One complication is that Jadud used the Java compiler's error message to classify the errors. These errors are not guaranteed to be stable across Java compiler versions, and thus we encountered several new messages in our data (e.g. "reached end of file while parsing") that are partial reclassifications of previous errors (e.g. "'}' expected"). This problem notwithstanding[2], our results, given in table 2, were broadly similar to those of Jadud.

Jadud's data had around 70,000 compiler errors to process, collected over two years. Our Blackbox data set, collected over nearly six months to 2013-12-01 (exclusive), has over 5,000,000 compiler errors. More generally, the larger sample sizes available in the Blackbox data will allow us to replicate and verify findings from previous smaller-scale studies.

[2]We hope to address this issue in our future work, by producing stable classifications of errors that are independent of compiler error messages.

```
System.out.println("Ingrese los numeros");
Scanner teclado = new Scanner(System.in);
int numero1 = teclado.nextInt();
int numero2 = teclado.nextInt();
if (numero1 > numero2) {
   // ...
}
if (numero1 < numero2); {
   // ...
}
```

Figure 1: An example of an empty if; note the semicolon on the final if.

6.2 Empty If Statements

When beginners learn Java, they must also learn the syntax. In our personal experience of teaching and delivering workshops, we have found that some beginners learn to put a semicolon at the end of *every* line they write. However, this can lead to a problem with constructs such as if-statements. For example, a student might write:

```
if (x < 5);
{
    x = 5;
}
```

The semicolon at the end of the first line is counted as a single empty statement, which acts as the body of the if-statement. This means that the first line has no effect, and the body (assigning 5 to x) is executed regardless. This error is particularly pernicious, because it is valid Java code and does not cause a compile error, so the user gets no explicit feedback about the problem. It also forms a useful test case, because it can be easily detected through automatic analysis.

We believe it is safe to assume that an if-statement, with no else clause, that has such an empty body is always a mistake by the programmer. The if-statement is redundant if there is no body and no else – and even if there is a side effect (e.g. if (x++ < 5);) then the same effect could be achieved with a normal statement (e.g. x++). We provide a real example from the Blackbox data in figure 1.

Our experience of the error is personal and anecdotal. We have two questions to investigate this error more rigorously:

1. How prevalent is this mistake?
2. How long does it take before the user fixes the mistake?

With the Blackbox data, the method is straightforward. We scan all successful compilations for the first occurrence of this mistake in a source file, then scan forwards in time, to see how many further successful compilations the user makes before this mistake is no longer present in the file.

Searching 1085598 different source files that had been successfully compiled at least once (average: 6.0 times), we found a total of 2647 source files containing these empty if statements. In 1513 cases, these empty ifs were later removed; the number of subsequent successful compilations that were performed before the empty if was removed are shown in figure 2. The results are clearly a power law, and most are immediately fixed, although note that some empty if statements were not fixed before more than 16 compilations. In the case of the other 1134 empty ifs: they were not fixed at all, and some were present for over 20 subsequent successful compilations.

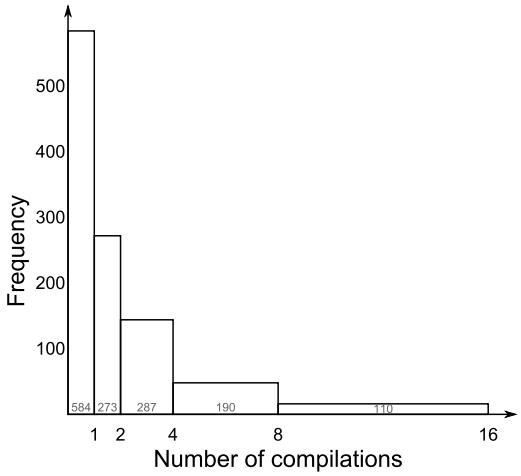

Figure 2: The number of successful compilations before an empty if was removed. For example, 584 empty ifs were removed during the next compilation (i.e. after 1 further successful compilation). A further 69 (not shown here) were removed after more than 16 further successful compilations.

This analysis demonstrates a small example of the sorts of scalable source code analysis that are possible on the Blackbox data set. More in-depth analyses could be performed that look for certain idioms or patterns in students' programs, or the use of certain programming constructs such as inheritance or recursion.

6.3 Tracking User Behaviour

Many students using BlueJ also use the BlueJ 'Objects First' textbook [3] for their course. The textbook is accompanied by a series of starting projects around which the book's exercises are based. By identifying this source code verbatim in the data, we can observe what the Blackbox users do with these projects. For example, the earliest code-writing exercises in the book are reproduced in figure 3, and the starting state of the source code is given in figure 4. We found 1491 users who opened this project and observed their next changes at subsequent successful compilations:

- 48 users changed line D (the wrong line) to use `"blue"` as the parameter, with 1 encountering a compiler error. 24 of these then made the same change to line B (the correct line).
- 503 users changed line B (the correct line) to use `"blue"` as the parameter, with 9 encountering a compiler error. Of these, 450 continued and:
  - 23 changed `"blue"` back to `"yellow"`, with 1 encountering a compiler error.
  - 67 added `"private Circle sun2;"` after line A, with 13 encountering a compiler error.
- 28 users added `"private Circle sun2;"` after line A, with 6 encountering a compiler error.
- 72 users added `"sun.slowMoveVertical(`*N*`);"` after line C, with 18 encountering a compiler error. 34 users chose 250 pixels as the parameter, and 16 users chose 300.

- **Exercise 1.16** In the source code of class `Picture`, find the part that actually draws the picture. Change it so that the sun will be blue rather than yellow.
- **Exercise 1.17** Add a second sun to the picture. To do this, pay attention to the field definitions close to the top of the class. You will find this code:

  ```
  private Square wall;
  private Square window;
  private Triangle roof;
  private Circle sun;
  ```

  You need to add a line here for the second sun. For example:

  ```
  private Circle sun2;
  ```

  Then write the appropriate code for creating the second sun.
- **Exercise 1.18** Challenge exercise... Add a sunset to the single-sun version of Picture. That is: make the sun go down slowly. Remember: The circle has a method `slowMoveVertical` that you can use to do this.

Figure 3: The first code-writing exercises from the BlueJ Objects First textbook, fifth edition [3]. The identical exercises appeared in previous editions, labelled 1.13–1.15.

```
public class Picture
{
    private Square wall;
    private Square window;
    private Triangle roof;
    private Circle sun; // A

    public void draw()
    {
        // ... initialise wall, window, roof

        sun = new Circle();
        sun.changeColor("yellow"); // B
        sun.moveHorizontal(180);
        sun.moveVertical(-(10));
        sun.changeSize(60);
        sun.makeVisible(); // C
    }

    public void setBlackAndWhite() { /* ... */ }

    public void setColor()
    {
        if (wall != null)
            wall.changeColor("red");
        window.changeColor("black");
        roof.changeColor("green");
        sun.changeColor("yellow"); // D
    }
}
```

Figure 4: A slightly condensed version of the original source code of the Picture class from the BlueJ Objects First textbook. Comments added to identify truncations and pertinent lines.

These changes were identified by transforming the source code into a canonical form (an abstract syntax tree) to eliminate whitespace differences, and then combining identical versions and ranking them by frequency. This analysis is simple, but gives an idea of some of the chronological traces that can be identified from the source code, and which may be useful, for example, for designing intelligent tutors (or for improving the flow of the BlueJ textbook!). Significant challenges remain in interpreting (and visualising) the results of such an analysis.

7. CONCLUSIONS

Blackbox is a perpetual data collection project. Its novelty lies in its size and in its availability to others. We gather data from hundreds of thousands BlueJ users all around the world into a single dataset, which other researchers can request access to. So far, researchers from eight other institutions have expressed interest and been provided with access to the data, and we welcome more[3]. Interested researchers should contact the authors for more information.

There remain several potential technical challenges in continuing the project. We must ensure that the server can withstand the load of recording all the data sent to it. We may have to manage changes in the data schema in future, should the need for changes arise. However, the project as it stands is ready for other researchers to access the data and start investigating their research questions.

Although researchers will each be investigating different specific questions, they are likely to share some common questions and requirements, both technical and research. Technically, researchers may need similar tools – for example, we have already created a tool that extracts all compilation inputs from the database and stores them in an easily accessible format. We intend to support the community of Blackbox researchers by providing means of communication (initially: a mailing list) and technical support. We hope that, over time, various researchers will create and share analysis and visualisation tools that ease the analysis of the data and allow others to perform investigations more easily. We also hope that joint discussions will serve to generate ideas and generate a common understanding about the potential and the limitations of the data.

There remain many technical challenges in the automated analysis of source code. The scale of the data does not make the analysis impossible, but it does prevent any manual intervention; any analysis must typically be completely automatic. Many research questions will likely also involve chronological analysis of source code, which adds further complexity. Researchers will likely need to look to tools such as source code query languages [10], and/or build new models to analyse source code development over time.

Although the Blackbox project is one of the first educational research projects to operate at such a large scale, it will surely not be the last. Sites such as MOOCs (Udacity, Coursera, etc), Khan Academy, Codecademy, Scratch 2.0 are already operating at a large scale with some or all data already stored server side (rather than having to be sent to a server specifically, as in BlueJ). For these projects, obtaining the source code data and IDE interactions of users will be even easier than the BlueJ project, and can operate at a similar (or even larger) scale. The MOOC sites may also have (and be willing to use) demographic information about their participants. Analysis techniques for source code are likely to be broadly applicable across several of these code bases, but analysis of IDE interactions may be less portable. It is also important to recognise that although large-scale data is useful for investigating some research questions, it is not a panacea. In some cases it is inappropriate and in other cases it will form a useful complement to other techniques, not replace them.

[3]We restrict access to researchers employed at established research institutions and with a record of interest in computer science education, to prevent misuse of the data. If the data were to become publicly accessible, Blackbox would constitute the largest public repository of programming assignment solutions for students.

8. REFERENCES

[1] *MSR '11: Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011.

[2] M. Ahmadzadeh, D. Elliman, and C. Higgins. An analysis of patterns of debugging among novice computer science students. In *ITiCSE '05*, pages 84–88. ACM, 2005.

[3] D. J. Barnes and M. Kölling. *Objects First with Java: A Practical Introduction using BlueJ*. Prentice Hall / Pearson Education, fifth edition, 2012.

[4] S. H. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola. Comparing effective and ineffective behaviors of student programmers. In *ICER '09*, pages 3–14. ACM, 2009.

[5] 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*, pages 296–300. ACM, 2009.

[6] M. C. Jadud. *An Exploration of Novice Compilation Behaviour in BlueJ*. PhD thesis, Computing Laboratory, University of Kent, January 2007.

[7] M. Kölling and I. Utting. Building an open, large-scale research data repository of initial programming student behaviour. In *SIGCSE '12*, pages 323–324. ACM, 2012.

[8] J. C. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy pascal programs. *Hum.-Comput. Interact.*, 1(2):163–207, June 1985.

[9] R. Thomas, G. E. Kennedy, S. Draper, R. Mancy, M. Crease, H. Evans, and P. Gray. Generic usage monitoring of programming students. In *ASCILITE '03*, 2003.

[10] R.-G. Urma and A. Mycroft. Programming language evolution via source code query languages. In *PLATEAU '12*, pages 35–38. ACM, 2012.

[11] I. Utting, N. C. C. Brown, M. Kölling, D. McCall, and P. Stevens. Web-scale data gathering with BlueJ. In *ICER '12*, pages 1–4. ACM, 2012.