Modeling and Simulation of Software Developer Behavior

Philippe Nguyen
McGill University
pnguye23@cs.mcgill.ca

Abstract

In this paper, we present a simulator for software developer behavior. The motivation behind this project is to model the behavior of a software developer conducting a program understanding task with a particular concern in mind. Applications for this system could be in research, where case studies could be made less expensive, or in the design of software development tools. As a whole, the simulator takes the shape of a non-deterministic decision generator. An important feature of the simulator is the modeling of two levels of developer expertise: beginner and advanced.

Validation was done on two medium sized systems: jHotDraw, and jEdit. From the test runs performed, we observe that the simulator behaves as expected. Limitations to the simulator include some performance issues, and the level of sophistication used in modeling developer behavior. Improvements to the system include better artificial intelligence, and added detail for the source model.

1. Introduction

Software engineering distinguishes itself from other engineering discipline by the importance human behavior holds in practice. This has led to an outburst of research on human behavior when performing software engineering and reengineering tasks. This type of research usually requires qualitative experiments involving live subjects. For example, one study involved observing developers of different levels of expertise as they investigated code in the aim of modifying a feature [1]. A common concern of such experiments is the time and effort spent in finding subjects, making them perform a task, and transcribing the qualitative data collected. Our project seeks to build a simulator modeling the behavior of a software developer performing a code navigation task. This work is based on M. Robillard's paper that describes an empirical study aimed at evaluating the efficiency of software developers in code investigation tasks [1].

The first part of this paper describes at a high level the functional properties of the simulator. The projected challenges are listed in the next section. The following part gives a description of the design decisions made to cope with these challenges. Next, we validate our tool on 2 source models. First, we test the simulator a source model of jHotDraw, a Java GUI framework for technical and structured graphics. Second, we apply the simulator to jEdit, a text editor implemented in Java representing a slightly larger system. Finally, we conclude with limitations of our approach and projected future work.

2. Functional Description

The simulator will try to replicate the behavior of software developers navigating code. Developers are assumed to have a goal in mind while traversing code.

The input of the simulator is the program description: it describes all classes, methods, fields, and relations of the code to be navigated, as well as the file structure of the system. In addition, the simulator has multiple parameters to consider:

1) Parameters of developer level: using previous studies, we can model the level of expertise of a developer. This will influence the model of developer behavior
2) Description of a concern that the developer should try to uncover as he/she navigates the code
3) Starting method: the method by which the developer can start navigating the code base

The output of a simulation run will be an investigation transcript, which, as described in [2], constitutes of a list of discovered methods, and the activity by which they were discovered. The possible activities are: B – opening a file from the code browser, such as the one in the Eclipse IDE; C – discovering an element using a cross-reference search; R – recalling an element previously opened, such as re-opening an editor window from a tabbed pane; L –

scrolling up and down in a file; K – a keyword search. Because of complexity reasons, our simulator does not currently support K activities.

3. Challenges

This project presents many challenges. One of them will be the level of smartness of the behavior model: in order to model a developer who has a defined goal in mind (i.e. to explore a specific concern), assigning probability on choices of action will ultimately have to be done in a smarter way than uniformly. Also, different ways of terminating the simulation will be explored. For example, we could take the approach of stopping the simulation when a certain number of events are captured; or, we could start with a target set of methods to be discovered, and stop the simulation when those methods are actually discovered by the simulator. In addition, determining how to assign the probabilities to each choice will be a difficult task: having only qualitative descriptions of developer behavior, we will have to translate them into quantitative parameters.

In a simulation point of view, the first challenge will be to have a clear definition of what represents the state. Secondly, we will need to determine a suitable level of abstraction, and a time base. For example, we could abstract timing information away and only work at a discrete time step level. In this situation, time is encoded only in the order of steps in the investigation transcript. Lastly, the core issue will be to implement the transition function, which essentially represents the developer's behavior when making new decisions.

4. Design

As a whole, the simulator will take the shape of a non-deterministic decision generator. Based on the present state of the simulator, the modeled developer is presented with a set of possible choices of methods and activities, from which the model will select its next decision. A large part of the simulator will then be to assign the proper probability to each possible choice in order to simulate as realistically as possible the thinking process of a developer who needs to investigate code in order to uncover a certain concern.

The next sections describe the different components of the simulator. Design decisions taken to overcome the identified challenges are exposed.

4.1 Simulator Architecture

The simulation is composed of two main parts: a model and a simulation kernel. The overall architecture of the simulator is depicted in figure 1.

Inputs

`referencesDB` contains referencing information about the target source code. Information such as method calls done by methods, and field declarations is included in this input.

`orederedDB` contains file structure information about the target source code. File content, file size, and method size (in terms of number of characters) is given by this input. Also, crucial information about the ordering in which files and methods appear in the code browser is available through this input.

`Concern description` defines the concern motivating the developer's navigation task. The description could be done in a natural language; however, because of implementation considerations, keyword based descriptions are preferable.

`Developer level` specifies what level of developer expertise the program should simulate. Developer level is modeled as a set of parameters, such as the size of the memory and the relative importance to give to each activity. The details of all developer parameters are discussed in the "Interface" section. Currently, the simulator offers three presets: random, beginner, and advanced. The random level represents a developer who gives equal weight to all choices presented. We implement this level in order to have a base against which to compare other levels.

4.2 Code Structure Model

The code structure model remains static, and is only queried by the developer model during simulation.

The code structure is modeled as a digraph, in which vertices represent the elements (classes, fields, and methods), and edges encodes a reachability relation between two elements. For example, element B is reachable from element A if there is a cross-reference relation from A to B, or if B is physically in proximity of A (e.g. the developer can reach B by shortly scrolling from A).

The model also considers a derived type of cross-reference: accesses by elements to a common field. Essentially, we model the fact that, even though two elements might not call each other directly, they still might be related if they access the same field.

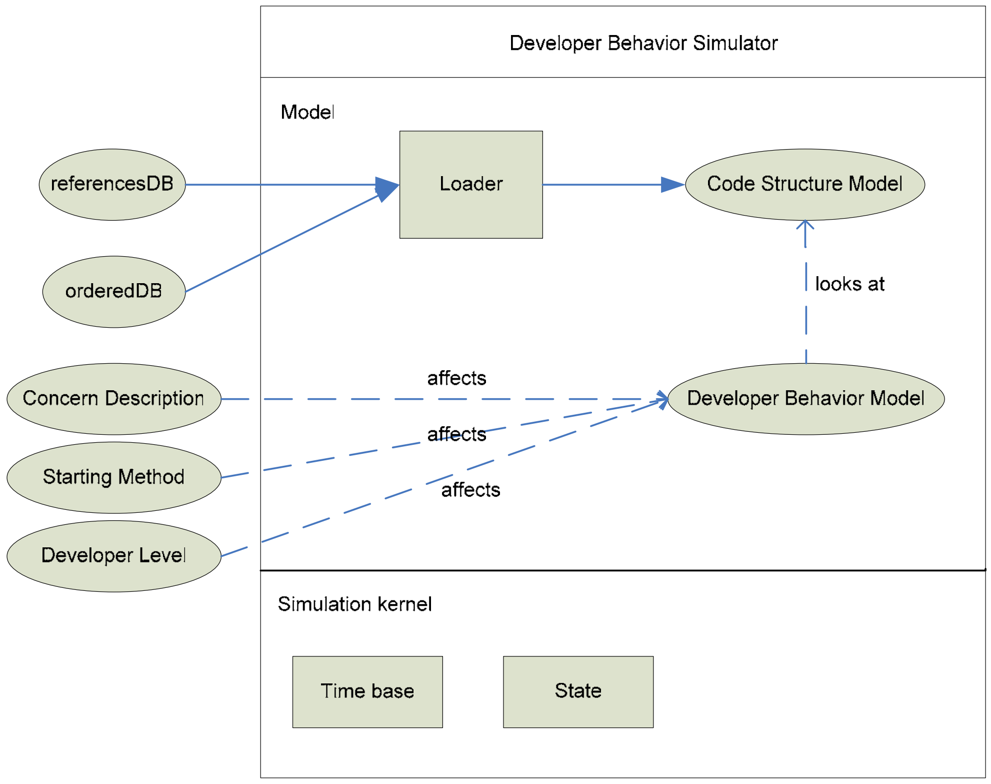


Figure 1 Architecture of the Software Developer Behavior Simulator

The reader should note that in the current version of the simulator only method elements are considered. This is for performance and complexity reasons. However, it is not far from reality, as developers mostly spend their time examining methods. Additionally, results presented in [1] also ignore fields.

The methods reachable from a B activity are the ones last viewed in the set of files that are in a certain range in the browser from the currently opened file.

The methods reachable from an R activity are the ones last viewed in the set of files previously opened. They are stored inside a buffer. For performance reasons again, the buffer size is limited, which, in reality, could hold true if the environment used in the navigation task has the option to limit the number of simultaneously opened tabs.

In order to save on space, references to `java` libraries are omitted from the code structure model. Even though we might believe that developers would seldom navigate through these libraries, this estimation does take away some realism from the simulator.

4.3 Developer Behavior Model

The developer behavior model looks at the code structure model in order to make decision. A decision is defined as a selected method to look at, using a certain activity.

Essentially, developer behavior is modeled as a choice generator followed by a semi-random decision maker: given a seed method, the developer queries the code structure model for other reachable methods, as well as the activities needed to reach them. Depending on the level of expertise, probabilities are assigned to each choice. Finally, a random number generator is called in order to select the next element to examine.

The transition function implements the process of generating the choices and making the decisions. It is affected by two things:

1) Developer level
2) Concern description

Developer Level

Based on [1], experts perform more C actions than L or B actions, since browsing and scrolling are considered less effective. Also, experts may not go back to what they've previously examined as often as beginners may. In other words, there will be smaller loops in a beginner's investigation trace. This is where the memory of the developer comes in: if developers remember having made a decision, they might give it less weight when selecting their next element to examine. The memory is modeled as a list of previously examined elements. To model the fact that beginners tend to refer back to previously examined elements, the maximum size of the memory is used as a differentiating parameter between advanced and beginner developers.

Concern description

The simulator should be able to take as input keywords describing the concern the developer would have in mind when traversing the code base. Thus, it needs a way to quantitatively correlate the concern description with the different methods in the code structure model. We achieve this by performing information retrieval (IR) techniques to determine which elements are more related to the concern at hand, and assign probability based on this.

In this case, the method names are parsed in order to build the index of keywords. Each method name represents a document, and the concern description is the query. The particular IR technique applied is the vector space model [2].

Implementation considerations

In order to keep the different influencing factors modularized, we employ a layered approach when assigning probabilities to choices.

First, based on the element being currently examined, generate the list choices (i.e. potential decisions consisting of {Element, Activity} tuples). The layers are applied to give weight to each choice:

Layer 1) Assign the same weight to all choices. The random developer level would stop here.

Layer 2) Factor-in the concern description. The weight of each choice is multiplied by some factor which takes into account the degree of similarity between the element and the concern description.

Layer 3) Factor-in the developer level. This mainly involves assigning more or less weight to a decision based on the activity it involves, and verifying whether the element of the choice is still in the developer's memory. When considering the memory, the simulator takes into account the "freshness" of the past decision: the fresher the choice's element is in the developer's memory, the less weight we should assign to that choice.

The weights are then normalized, and the cumulative distribution function is built. A pseudo-random number generator then selects from the CDF the next element to be examined.

This approach is advantageous because it keeps the number of if statements relatively controllable. It modularizes the influencing factors, so that they can easily be modified, removed, or added. Moreover, it allows users to enable or disable certain influencers.

Output function

The output function takes care of outputting the correct events in investigation transcript format. The format is described in [3].

4.4 Simulation Kernel

The simulation kernel is responsible for two things: the time-base, and updating the state of the developer behavior model.

Time base

The simulator's time base is discrete. The ordering of the events in the investigation transcript encodes the time progression of the simulator.

The number of iterations can be adjusted using the interface of the simulator. We decided not to opt for the option of setting a target set of goal methods to determine when to terminate the simulation because that approach would potentially create infinite simulations. Also, it does not simulate reality well: developers do not pre-define a set of target methods to explore. Rather, they investigate the code until they reach a certain level of confidence that they have understood the concern at hand.

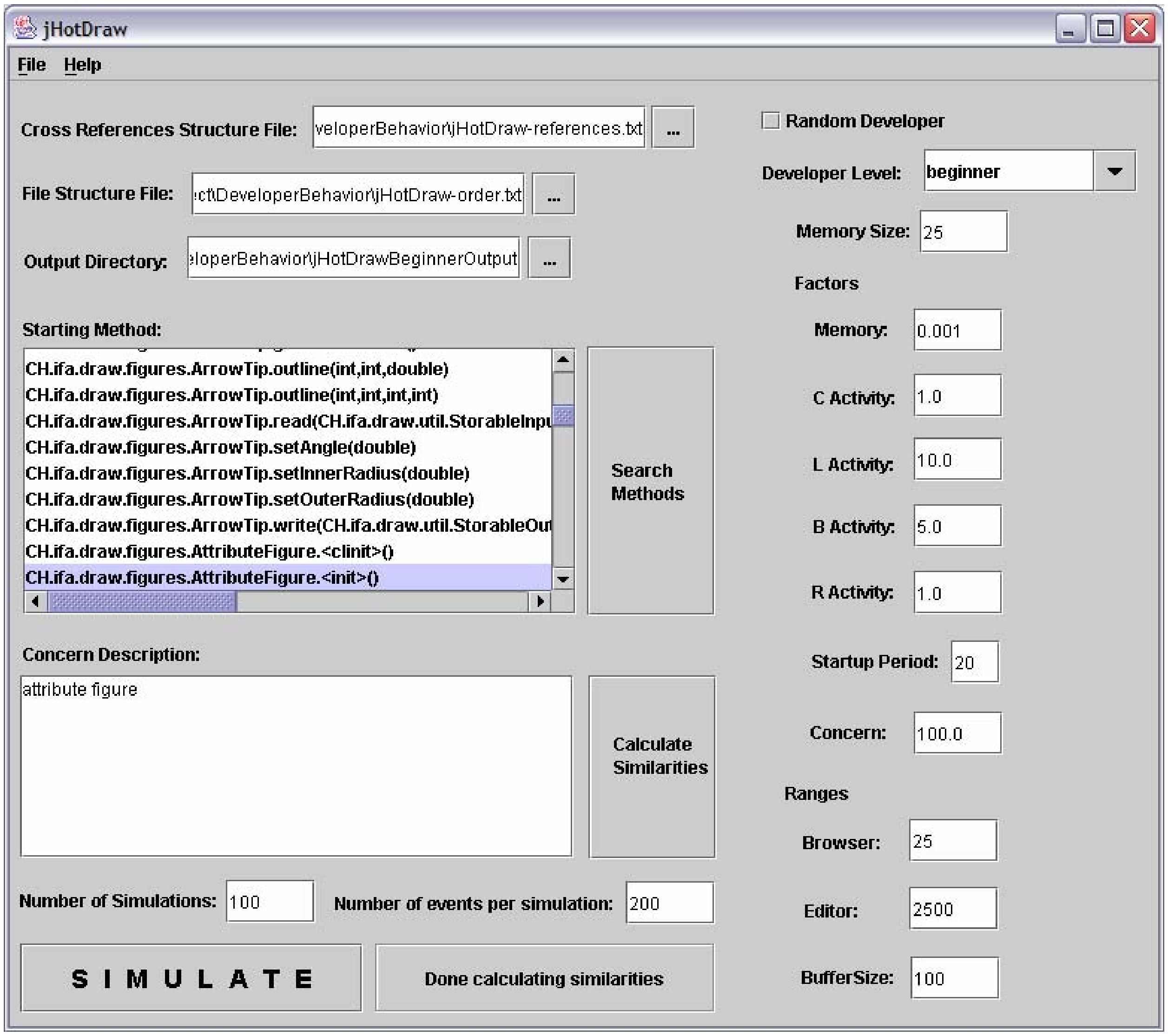
On the other hand, fixing a number of iterations is by no means the best approximation of reality: developers do not have a limit on the number of actions they can perform. Also, we cannot compare performance between levels of expertise in terms of time to complete a task. Still, fixing the number of iterations is the simplest approach that can at least ensure termination of the simulation. Implementing a better termination condition is one of our prominent future goals.

State

The state of the simulator is encapsulated inside the developer behavior model. It includes:

1) The current element examined
2) The contents of the accessed files buffer
3) The contents of the developer memory.

Updating the state consists of making a new decision, which translates to selecting a new element to examine, adding the containing file to the file buffer if a new file was opened, and adding the new decision in the memory.



Cross References Structure File: veloperBehavior\jHotDraw-references.txt

File Structure File: ct\DeveloperBehavior\jHotDraw-order.txt

Output Directory: eloperBehavior\jHotDrawBeginnerOutput

Starting Method:

CH.ifa.draw.figures.ArrowTip.outline(int,int,double)
CH.ifa.draw.figures.ArrowTip.outline(int,int,int,int)
CH.ifa.draw.figures.ArrowTip.read(CH.ifa.draw.util.StorableInpu
CH.ifa.draw.figures.ArrowTip.setAngle(double)
CH.ifa.draw.figures.ArrowTip.setInnerRadius(double)
CH.ifa.draw.figures.ArrowTip.setOuterRadius(double)
CH.ifa.draw.figures.ArrowTip.write(CH.ifa.draw.util.StorableOut
CH.ifa.draw.figures.AttributeFigure.<clinit>()
CH.ifa.draw.figures.AttributeFigure.<init>()

Concern Description: attribute figure

Number of Simulations: 100

Number of events per simulation: 200

Random Developer

Developer Level: beginner

Memory Size: 25

Memory: 0.001

C Activity: 1.0

L Activity: 10.0

B Activity: 5.0

R Activity: 1.0

Startup Period: 20

Concern: 100.0

Browser: 25

Editor: 2500

BufferSize: 100

Figure 2 GUI of the Developer Behavior Simulator

4.5 Interface

Figure 2 shows the simulator's graphical user interface. On the left hand side, the user can specify the file paths of the `referenceDB` file and the `orderedDB` file, as well as specify the starting method and the concern description. Other simulation input can be entered, such as the number of simulations to run, and the number of desired events within each transcript. In order to allow experimenters to adjust the different parameters that influence the simulation of a developer, the right hand side of the window lets the user model the level of the developer, as well as adjust the properties of the development environment (IDE).

Developer Parameters

Random Developer Checkbox – checking this box puts the simulator in random mode, where all generated choices are weighted equally when making a decision.

Developer Level Combo Box – allows the user to select a saved parameter setting. Saving a setting (using the "Save Developer As" menu item) makes the setting available in the combo box.

Memory Size – sets the size of the developer's memory. Since the simulator takes into account the fact that developers might give less weight to elements that are still in the developer's memory, the size of the memory is an important factor in determining the level of expertise of the developer.

Memory Factor – the factor by which to multiply the weight of a choice when that choice is found to be in the memory of the developer. It should be between 0.0 and 1.0.

C, B, L, and R Factors – the factors by which to multiply the weight of a choice, based on the activity that the choice involves.

Startup period – sets a period of time at the beginning of the simulation during which the C, B, L, and R factors are ignored. In other words, it models the fact that, initially, developers may not give preference to any particular activity, and instead may give equal weight to all activities in order to arrive at an element that relates to the concern in mind as early as possible.

Concern Factor – the factor by which to multiply the weight of a choice when the method name of that choice has been determined to be related to the concern description. Setting this factor to a very high value will make the weights of choices related to the concern much larger than the weights of choices that are not related to the concern.

*IDE Parameter*s

Browser Range – specifies how many files are reachable above and below the file containing the element being currently examined. In other terms, it specifies how many elements are reachable by a B activity. This models the fact that developers may stay within a certain range when exploring in the browser.

Editor Range – specifies, in number of characters, how far a developer may scroll up and down within the code editor. In other terms, it specifies how many elements are reachable from an L activity: for the currently examined element, E, every other element contained inside the same file as E and for which the distance, in number of characters, from E is lesser than or equal to the editor range is considered reachable using by scrolling.

Buffer Size – sets the maximum size of the accessed files buffer. It models the fact that in certain IDEs, such as Eclipse, the developer can set the maximum number of opened panes. When the buffer has reached maximum capacity and a new file is opened, the pane that was opened earliest is closed.

5. Evaluation

In order to validate our approach, we conducted simulation runs on two real-world systems: jHotDraw and jEdit. The input files for each of these programs were provided by our supervisor. Our expectations from this experiment are:

1) That the beginner and advanced levels do differ from the random level: thus, we expect elements for the beginner and advanced levels to be more related to the described concern;
2) That the resulting traces will reflect the desired differences between a beginner and an advanced developer, namely that a trace for the advanced level will contain more C activities, and that the breadth of elements examined will be wider for the advanced developer.

In order to verify these hypotheses, we will mainly consider three aspects: the most frequently examined elements, the total number of elements examined, and the usage frequency of activities.

For each developer level, we ran 100 simulations and compiled the traces into a single file. Collected data include the consolidated frequency of each examined elements, the total number of elements observed, and the frequency for each type of activity. Tables 1 to 5 summarize the conditions under which the simulations were run.

Table 1: Resource Specification

CPU	AMD Athlon™ 64
Memory	512MB DDR RAM
Operating System	MS Windows XP Pro
Running Environment	Eclipse 3.0.1
JDK	1.4

Table 2: Developer Parameters Settings

	Beginner	**Advanced**
Memory Size	10	50
Memory Factor	0.001	0.001
C Factor	1.0	10.0
L Factor	10.0	2.0
B Factor	5.0	1.0
R Factor	1.0	1.0
Startup Period	20	20
Concern Factor	1000.0	1000.0

Table 3: IDE Parameters Settings

Browser Range	10 files
Editor Range	5000 chars
Buffer Size	20 panes

Table 4: Simulation Settings

Number of simulations	100
Number of events per simulation	200

Table 5: System-specific Inputs

jHotDraw

referenceDB	Available at on the project's website [4]
orderedDB	
Starting Method	`CH.ifa.draw.figures.Attribute.<init>()`
Concern Description	"attribute figure"

jEdit

referenceDB	Available at on the project's website [4]
orderedDB	
Starting Method	`org.gjt.sp.jedit.options.LoadSaveOptionPane._init()`
Concern Description	"autosave"

To reinforce our evaluation, we asked a developer possessing a thorough knowledge of both target systems' source code to examine investigation transcripts given by the simulator. Note that the choice of concern descriptions and seed methods for our simulation runs was suggested by this expert. This person's opinion on whether the simulator has generated realistic output is a determining factor in evaluating the usability of the system for future researchers.

5.1 Results

An aggregated version of the results from the simulation runs is presented in the appendices. The raw data is available on the project's website [4].

The results are excerpts from the tally files for jHotDraw and jEdit. These files are generated after each batch of simulations is executed. In this case, they are compilations of the 100 simulations of each developer level on each code base. The information contained includes: a list of the elements examined during all the simulations, sorted in descending order of frequency; the total number of examined elements; the frequency distribution of the activities used by the developer; and information about the developer and IDE parameters used during the simulations.

5.2 Analysis

To analyze the gathered data, we will first look at whether the output for the advanced and beginner levels presents significant differences from the purely random simulations. Next, we will compare the output from the advanced and beginner levels in order to verify our previously stated expectations.

5.2.1 Difference from Random Decision Making

Looking at the most frequently examined elements from both jEdit and jHotDraw, we see that the non-random levels give more importance to elements that are related to the concern. From this, we conclude that the IR module works as expected, and that a significant degree of difference from a purely random approach is achieved by our model of developer behavior.

Surprisingly, the distributions of activities used do not show a uniform distribution for the random simulations. This is probably a result of the nature of the different code structures: different structures may offer more or less opportunities for each activity. For example, a code base containing very long methods that cross-reference themselves often would present more opportunities for C actions, and fewer opportunities for L actions. Another surprise is that, in the case of jEdit, the total number of examined elements is larger for the advanced level than it is for the random level. Intuitively, we would expect a random simulation to cover more elements than a simulation in which mostly elements that are related to a concern are chosen. However, the memory factor involved in non-random simulations does have an effect of the breadth of the investigation transcripts; in fact, conducting a test run with a modified version of the advanced settings in which the memory factor is set to 1.0 yielded a total number of examined elements much lesser than the one for the random simulation.

5.2.2 Advanced vs. Beginner

The advanced and beginner levels differ in all observed aspects. The activity distributions reflect the activity factors set in the developer parameters for each level, with some previously discussed fluctuations due to code structure: C activities are predominant with the advanced developer, while beginner developers mostly use L and B activities. This concords well with the observations made in [1]. The IDE parameters also influence the distribution of the used activities. A larger browser range increases the number of potential decisions involving B activities, in which case a higher frequency of B activities would be observed. Varying the editor range has a same effect on L activities, and R activities are similarly affected by the buffer size.

Comparing the number of examined elements in both jHotDraw and jEdit reveals that indeed the breadth of examination for the advanced developer is much wider than the one for the beginner developer.

As discussed in the previous section, this is mostly due to the memory factor: larger memory size for advanced developers results in fewer repeated examination of the same element, and hence to more explored elements.

Looking at the elements that were most frequently examined, we observe that simulations for both levels of expertise mostly looked at the same elements. This is to be expected, since the location of the concern being explored does not change depending on the level of the developer. However, the frequency distributions for the advanced developer are much more uniform than for the beginner developer. Indeed, for the beginner level, much of the distribution is concentrated in the first three or four elements, while the distribution is more evenly spread out for the advanced level. This demonstrates that the simulator models the beginner developers' tendency to focus only on a few methods when navigating through code; contrastingly, advanced developers cover much more of the code base, and thus gain a broader understanding of the concern.

Also, in the case of jHotDraw, the fact that the top ten most examined elements for the beginner developer were all from the same package, whereas the top ten elements for the advanced developer were from diverse packages, reinforces our observation that beginners tend to localize their traversal in a much smaller area of the source code. At first sight, the reader might think that the advanced developer has missed some methods from the `AttributeFigure` package. However, looking at the complete results reveals that, in fact, the other methods of the package barely missed being included in the top ten elements.

5.2.3 Expert Opinion

The opinion of the person familiar with the jHotDraw and jEdit code bases on the realism of the simulator output is mitigated. On the positive side, he acknowledges that the most frequently examined methods for both systems are in fact very relevant to the concern descriptions. Also, he has recognized very realistic episodes inside the investigation transcripts. On the negative side, the results are still too random in his opinion. Additionally, he has recommended that class hierarchies be taken into consideration when assigning weights to choices. Moreover, according to him, the frequency of B activities is too high: as a remedy to this, we might think of setting a threshold based on similarity to the concern: elements in browser range that are not sufficiently relevant to the concern would not be included in the list of choices. Lastly, the expert has noticed an important deviation from reality presented by the simulator. Scrolling events don't seem to follow the order of the methods contained in the `orderedDB` file. Upon reflection, we realize that this problem is due to the fact that though distance from the currently examined element is taken into account when selecting which methods are reachable from an L activity, the actual value of the distance is not used when calculating the weight to assign to the choices. In other words, the simulator does not currently assign more weight to a method that is closer to the current one being examined.

6. Limitations

The simulator presents limitations other than the ones previously mentioned. An important one involves performance and scalability. While it probably still presents a speed-up as compared to actual experimentation with live participants, the simulator is not optimized performance-wise. The biggest hits lie in the loading of the code structure and the in the calculations done by the IR algorithm. However, once these two operations are done, the actual simulation runs in reasonable time. As an indication, jHotDraw required around 3 minutes to load into memory, while jEdit required around 10 minutes. Once loaded though, their simulations only took a few seconds.

Another limitation is the level of sophistication used to model human behavior. For example, the factors affecting the weights of each choice were chosen on a trial-and-adjust basis. More time would be needed to either tweak these factors in order to reflect reality as much as possible, or to apply other layers of influencers to the simulator.

7. Future Work

The simulator could benefit from more advanced artificial intelligence. In its current state, it does little to model human reasoning in depth. More work could be done to add more sophistication to the developer behavior model.

Future work could also include integrating a smarter termination condition. One possibility would be to specify a target confidence level. This could somehow measure the progress made by the developer in discovering elements related to the concern in mind. When enough confidence is obtained, the simulation would stop. Such an approach could potentially allow experimenters to determine if, for example, advanced developers finish their tasks in fewer actions than beginner developers. However, using a confidence level could still potentially make the simulator run infinitely; therefore, the approach would most probably need a failsafe, which could take the form of a limit in the number of events it can generate.

Adding detail to the code structure model could also make the simulator give more realistic output. Examples of potential details to be added are fields. Fields present a particular challenge in observational studies: experimenters using video captures to observe participant behavior cannot easily determine when a developer is examining a particular field, since many fields could appear at the same time on the screen. The observers would then have to explicitly ask the participants to explain which fields they are examining (method that is often referred to as “think aloud” process). This approach is however more time consuming and presents the danger of undesirable biasing of participant behavior. Therefore, integrating fields in the simulator would be a great advantage.

The realism of the simulator’s output would also require more validation. For this, more systems need to be tested upon.

8. Conclusion

The motivation for this project was to develop a simulator that would model the behavior of a developer who is trying to locate a concern inside a code base. Such a simulator could be used by researchers who study human-computer interactions, or by software engineers building software development tools.

The simulator keeps a model of the code structure and simulates developer behavior by making decisions on which elements to examine, and on which activities to perform. Developer level can be adjusted by specifying values for certain parameters, such as the size of the developer’s memory and the importance given to elements related to the concern.

Validation was performed on two real-world systems, jHotDraw and jEdit. The goals of the experiment were to see whether the implemented developer levels significantly differentiated themselves from a purely random simulator, to compare the advanced level with the beginner level, and to evaluate the degree of realism of the simulator’s output.

In all, the simulator presents great potential for automatically locating concerns using information retrieval techniques, and for generating investigation transcripts that model different levels of developer expertise. However, there are some limitations, such as scalability and performance, and sophistication of the artificial intelligence used. Future work for this project should thus be aimed at improving those aspects.

9. References

[1] M.P. Robillard, W. Coelho, and G.C. Murphy, *How Effective Developers Investigate Source Code: An Exploratory Study*, IEEE Transactions on Software Engineering, November 2004

[2] M.P Robillard and G.C Murphy, *Automatically Inferring Concern Code from Program Investigation Activities*, Proceedings of the 18th International Conference on Automated Software Engineering, pp. 225–234, IEEE Computer Society Press, October 2003

[3] Rich Ackerman, “Vector Model Information Retrieval”, *Theory of Information Retrieval*, Florida State University, September 2003, URL: http://www.hray.com/5264/math.htm

[4] Project Website, URL: http://www.cs.mcgill.ca/ ~pnguye23/COMP522/project/project.htm

APPENDIX A - Results for jHotDraw

Top 10 most frequently examined elements

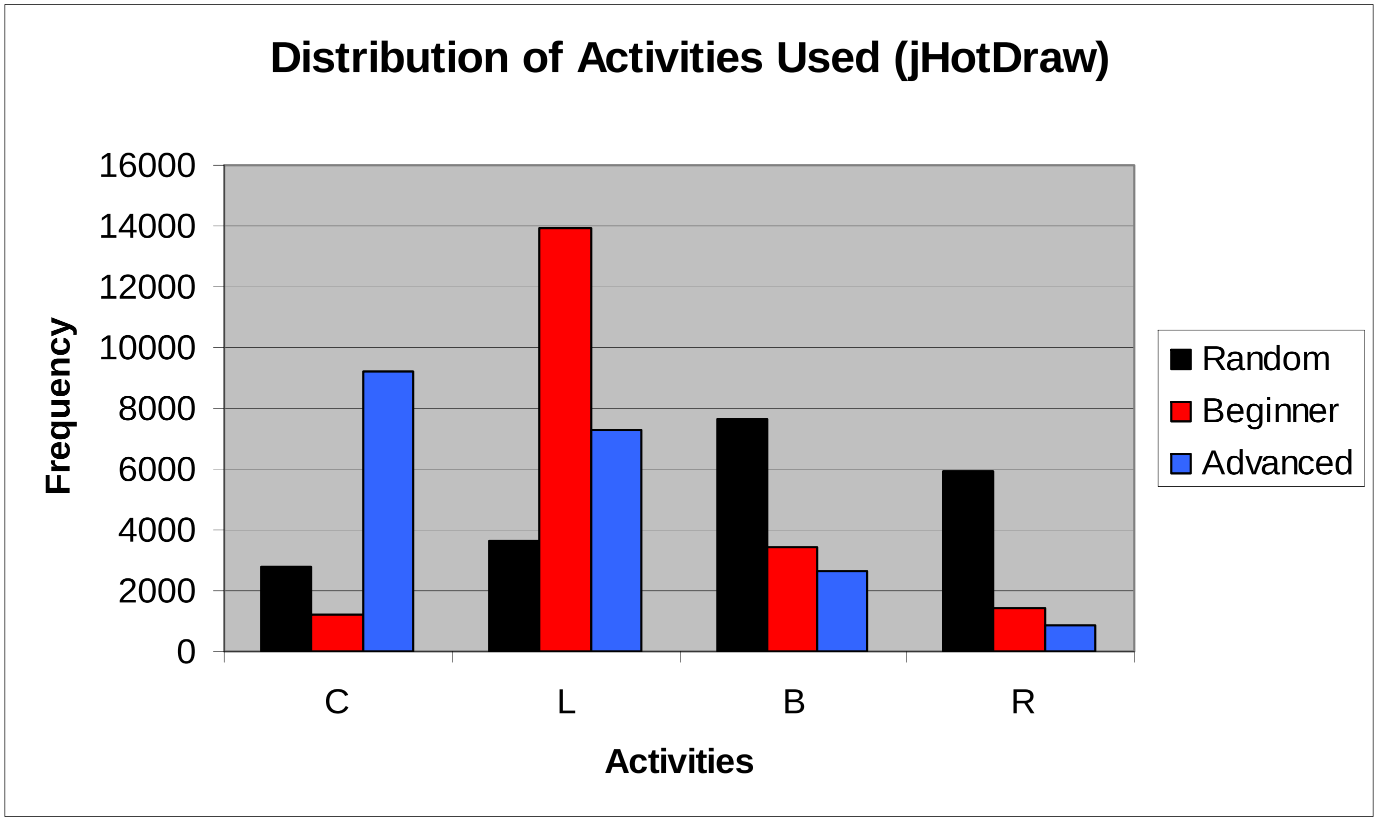
Random		Beginner		Advanced	
Element	Freq	Element	Freq	Element	Freq
`CH.ifa.draw.figures.AttributeFigure.<init>()`	206	`CH.ifa.draw.figures.AttributeFigure.<init>()`	623	`CH.ifa.draw.figures.AttributeFigure.getAttribute(java.lang.String)`	247
`CH.ifa.draw.samples.javadraw.JavaDrawApp.createTools(javax.swing.JToolBar)`	136	`CH.ifa.draw.figures.AttributeFigure.getAttribute(java.lang.String)`	607	`CH.ifa.draw.framework.Figure.setAttribute(java.lang.String, java.lang.Object)`	243
`CH.ifa.draw.samples.javadraw.JavaDrawApplet.createTools(javax.swing.JPanel)`	128	`CH.ifa.draw.figures.AttributeFigure.getDefaultAttribute(java.lang.String)`	543	`CH.ifa.draw.applet.DrawApplet.createAttributeChoices(javax.swing.JPanel)`	237
`CH.ifa.draw.figures.LineDecoration.draw(java.awt.Graphics,int,int,int, int)`	128	`CH.ifa.draw.figures.AttributeFigure.setAttribute(java.lang.String,java.lang.Object)`	533	`CH.ifa.draw.figures.PolyLineFigure.setAttribute(java.lang.String, java.lang.Object)`	236
`CH.ifa.draw.figures.GroupFigure.handles()`	116	`CH.ifa.draw.figures.AttributeFigure.draw(java.awt.Graphics)`	519	`CH.ifa.draw.standard.ChangeAttributeCommand.<init>(java.lang.String,java.lang.String, java.lang.Object, CH.ifa.draw.framework.DrawingEditor)`	224
`CH.ifa.draw.samples.nothing.NothingApplet.createTools(javax.swing.JPanel)`	110	`CH.ifa.draw.figures.AttributeFigure.read(CH.ifa.draw.util.StorableInput)`	491	`CH.ifa.draw.figures.AttributeFigure.setAttribute(java.lang.String, java.lang.Object)`	224
`CH.ifa.draw.figures.GroupHandle.<init>(CH.ifa.draw.framework.Figure, CH.ifa.draw.framework.Locator)`	107	`CH.ifa.draw.figures.AttributeFigure.initializeAttributes()`	485	`CH.ifa.draw.standard.ChangeAttributeCommand.execute()`	220
`CH.ifa.draw.figures.ElbowHandle.<init>(CH.ifa.draw.figures.LineConnection, int)`	96	`CH.ifa.draw.figures.AttributeFigure.drawFrame(java.awt.Graphics)`	477	`CH.ifa.draw.framework.Figure.getAttribute(java.lang.String)`	220
`CH.ifa.draw.samples.pert.PertApplet.createTools(javax.swing.JPanel)`	96	`CH.ifa.draw.figures.AttributeFigure.drawBackground(java.awt.Graphics)`	475	`CH.ifa.draw.figures.GroupFigure.setAttribute(java.lang.String, java.lang.Object)`	210
`CH.ifa.draw.figures.ShortestDistanceConnector.findPoint(CH.ifa.draw.framework.ConnectionFigure, boolean)`	93	`CH.ifa.draw.figures.AttributeFigure.getFrameColor()`	465	`CH.ifa.draw.standard.ChangeAttributeCommand.createUndoActivity()`	205

Total number of elements examined

	Random	Beginner	Advanced
Number of elements examined	1866	749	924

Frequency distribution of activities

Activity	Random	Beginner	Advanced
C	2772	1222	9192
L	3633	13909	7265
B	7661	3422	2655
R	5934	1447	888



APPENDIX B - Results for jEdit

Top 10 most frequently examined elements

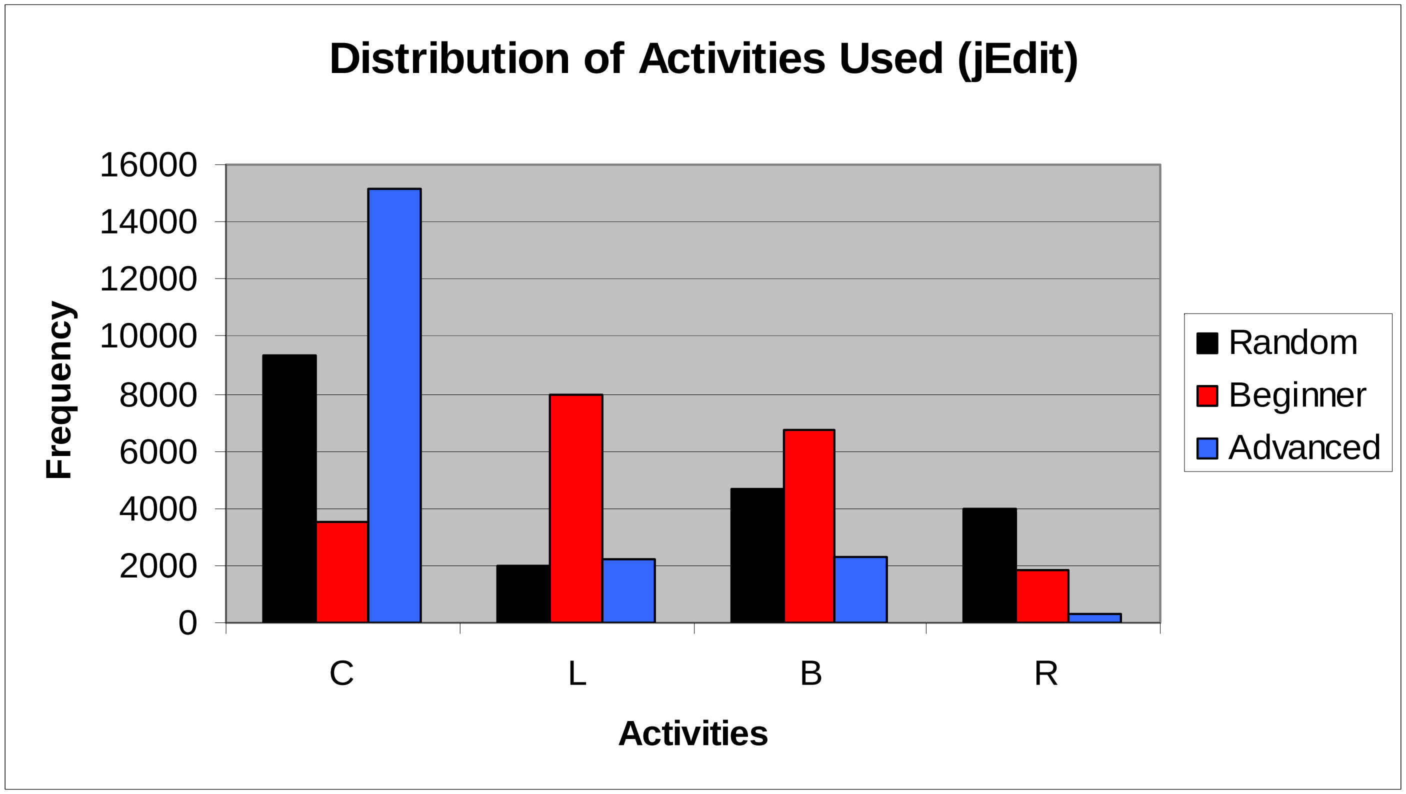
Random		Beginner		Advanced	
Element	Freq	Element	Freq	Element	Freq
`org.gjt.sp.jedit.options. LoadSaveOptionPane._init()`	227	`org.gjt.sp.jedit.Autosave. actionPerformed (java.awt.event.ActionEvent)`	1417	`org.gjt.sp.jedit.Autosave. actionPerformed (java.awt.event.ActionEvent)`	402
`org.gjt.sp.jedit.textarea. JEditTextArea.scrollTo (int,int,boolean)`	223	`org.gjt.sp.jedit.Autosave. stop()`	1295	`org.gjt.sp.jedit.Buffer. autosave()`	292
`org.gjt.sp.jedit.jEdit. getProperty(java.lang.String)`	185	`org.gjt.sp.jedit.Autosave. setInterval(int)`	1290	`org.gjt.sp.jedit.Autosave. setInterval(int)`	266
`bsh.BSHArrayDimensions.eval (bsh.CallStack,bsh.Interpreter)`	179	`org.gjt.sp.jedit.Autosave. <init>()`	1290	`org.gjt.sp.jedit.Autosave. stop()`	264
`gnu.regexp.RETokenChar.chain (gnu.regexp.REToken)`	165	`org.gjt.sp.jedit.Buffer. autosave()`	983	`org.gjt.sp.jedit.Autosave. <init>()`	224
`org.gjt.sp.jedit.Registers. setRegister(char,java.lang.String)`	120	`org.gjt.sp.jedit.Buffer. getAutosaveFile()`	310	`org.gjt.sp.jedit.Buffer. getAutosaveFile()`	172
`org.gjt.sp.jedit.options. LoadSaveOptionPane._save()`	116	`org.gjt.sp.jedit.Buffer. recoverAutosave (org.gjt.sp.jedit.View)`	138	`org.gjt.sp.jedit.Buffer. recoverAutosave (org.gjt.sp.jedit.View)`	163
`bsh.ParseException.getErrorText()`	109	`org.gjt.sp.jedit.options. LoadSaveOptionPane._init()`	136	`bsh.BSHArrayDimensions.eval (bsh.CallStack,bsh.Interpreter)`	145
`org.gjt.sp.jedit.pluginmgr. PluginManager.updateTree()`	79	`org.gjt.sp.jedit.SettingsReloader. maybeReload(java.lang.String)`	110	`org.gjt.sp.jedit.textarea. JEditTextArea.scrollTo (int,int,boolean)`	127
`org.gjt.sp.jedit.options. GutterOptionPane._init()`	78	`org.gjt.sp.jedit.buffer. BufferIORequest.autosave()`	82	`org.gjt.sp.jedit.options. LoadSaveOptionPane._init()`	116

Total number of elements examined

	Random	Beginner	Advanced
Number of elements examined	2628	1993	3022

Frequency distribution of activities

Activity	Random	Beginner	Advanced
C	9334	3505	15163
L	1971	7936	2194
B	4703	6700	2329
R	3992	1859	314


Distribution of Activities Used (jEdit)
Frequency
C
L
B
R
Activities
Random
Beginner
Advanced