Task-specific source code dependency investigation

Reid Holmes and Robert J. Walker
Laboratory for Software Modification Research
Department of Computer Science
University of Calgary
{rtholmes, rwalker}@cpsc.ucalgary.ca

Abstract

We present a simple, visual approach to help developers view and navigate structural dependency information, aimed specifically at pragmatic reuse tasks. Our visual approach, implemented as the Gilligan tool, leverages standard GUI widgets (such as lists and editors) that developers are adept at using. Gilligan represents complex dependency data in a simplified format, appropriate for investigating reuse tasks. We present a small-scale, semi-controlled experiment that indicates that the approach permits more accurate identification of relevant structural dependencies with a lower time investment, as compared to traditional manual approaches. Last, we discuss the potential for the approach to aid in other specific software understanding tasks.

1. Introduction

Developers often wish to reuse source code in ways that it has not been designed to be reused. Developers undertaking such *pragmatic reuse tasks* can benefit from tool support to quickly and accurately identify the structural dependencies (e.g., classes and methods that they reference) of any code fragments they are considering reusing. In planning these tasks, developers need a specific subset of information encoded in the source code they want to reuse. As such, general graph visualization techniques are not needed; indeed, the general nature of full-blown graph visualization can inhibit the understanding needed for dependency analyses for pragmatic software reuse [4].

Understanding the scope of a fragment's dependencies is essential for a developer to make an informed decision about whether to reuse the fragment of source code or to re-implement its functionality [3]. After the developer finds the core of the functionality that she is interested in reusing, she must consider how best to "triage" its dependencies on the rest of the original system. To do this, she traces each dependency from that core to the entity being referenced (e.g., a type, method, or field). If the referenced entity also provides useful functionality, the developer is likely to add it to the code to be reused. If the referenced entity is of no use to the developer, she then has to consider whether the dependency can be dead-ended (i.e., the method call or field reference will be eliminated), or remapped to a different entity in her target system. Ultimately, she wants to minimize the unwanted functionality that will be incorporated into her system. We refer to the end-product, that is the collection of nodes and triage decisions, of this investigation process as a *pragmatic reuse plan*.

We consider there to be three key properties needed to support this investigation and decision process: *propagational navigation*, where a source code entity need be investigated only if it is reachable transitively through other entities that are to be reused; *low commitment*, where early decisions can be changed and the navigation can be easily continued as needed; and *flexibility in abstraction*, where visualization at any level between the package level to the detailed source code might be needed [8].

Current integrated development environments (IDEs)—such as Eclipse, IntelliJ, or Visual Studio—facilitate the navigation of the structural dependencies of source code. However, IDEs focus on the source code itself, provide limited views of the dependencies (typically one file at a time), and do not help the developer to enumerate these dependencies and track their decisions about how they should be triaged.

We have developed a source code dependency visualization mechanism specifically for developers who are investigating and planning pragmatic reuse tasks. Our approach aims to better support developers performing these tasks by providing only the subset of features they need [6]. While using a general graph-based visualization seems like a natural fit for this task, these techniques fail to adequately support propagational navigation, quickly leading the developer to be disoriented by a proliferation of relationships and entities [2, 3]. Our primary concern is to reduce the cognitive effort required of the developer while investigating the structural dependencies for a source code fragment.

By focusing on the three key properties (propagational navigation, low commitment, and flexibility in abstraction), our tool can be more focussed as compared to more generic techniques. In addition, our tool (called Gilligan) provides an interface based on standard GUI widgets (tree lists and editors) thereby minimizing the presence of unfamiliar interaction techniques. One key aspect of our approach is that it allows developers to record their decisions as they investigate individual dependencies; this aims to reduce their cognitive burden by relieving them of having to remember every decision they have made.

We evaluate our approach via a semi-controlled experiment that measured the ability of six developers to determine the structural dependencies of four different source code fragments using either our tool or standard IDE tools. Our investigation focuses on the ability of our visualization technique to help developers identify all of the structural relationships within a fragment of source code. We found developers were able to more completely identify these structural relationships in less time using our tool than with standard manual approaches.

The remainder of the paper is structured as follows. Section 2 describes a motivational scenario involving a pragmatic reuse task. Section 3 details the design requirements of our approach to support the needs of developers performing pragmatic reuse tasks and is followed by related work in Section 4. Our visualization approach is outlined in Section 5 and the evaluation follows in Section 6. Section 7 discusses remaining issues.

This paper contributes a visual approach that helps developers to quickly identify and triage both the direct and indirect structural dependencies of any source code fragment. This mechanism is evaluated through a small-scale, semi-controlled experiment measuring its effectiveness compared to standard developer practice.

2. Motivating Scenario

Consider a developer who wants to create a system that listens for events using simple sockets. From her experience, she knows that Log4J[1] has the ability to send events via sockets. She also knows of Ganymede[2], a system that provides a Log4J view within the Eclipse IDE that receives its events through sockets. As Ganymede does a task similar to what she wants to provide, she investigates this project to see if she can reuse any of its source code; while Ganymede was not designed for reuse, it still may offer functionality she can benefit from.

While investigating the dependencies of this pragmatic reuse task, she is essentially interested in answering one question: "Where in the source code is a feasible boundary for the feature I would like to reuse?" To answer this question, she would like to know the extent of each dependency of the code she is considering reusing. By knowing the scope she can determine if a dependency is trivial (for instance, being dependent on `java.lang.String` is not a problem for Java programs) or onerous (for instance, if the program was dependent on a method that had 1000 other dependencies of its own). She can use this knowledge to determine where to create the boundary of the feature.

After a quick search though the Ganymede source code, she locates the relatively simple `Log4JServer.startListener()` method. This method is only 18 lines long and contains 15 method calls, 2 object instantiations, 1 field reference, and 1 local variable assignment. At a glance, she can see that `startListener()` does exactly what she wants. What she cannot tell at a glance is how tangled `startListener()` is with the rest of Ganymede. Investigating each of the dependencies using her IDE, the developer navigates through 22 methods spread throughout 15 classes in 9 packages. These packages include a range of functionality (including `java.net`, `org.eclipse.ui`, and `net.sf.ganymede.preferences`).

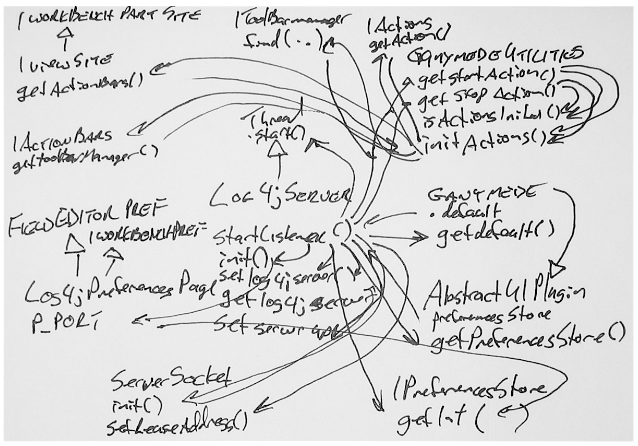


Figure 1. Manually created depiction of dependencies for `startListener()`.

If the developer decides to invest the time to research the extended dependency list, she will likely derive a picture similar to Figure 1. This picture has its drawbacks: It is not amenable to updates, nor can the developer see how it is directly associated to the source code. Also, if she wants additional information about any node in the diagram, she must locate and open that type in her IDE and check manually. These shortcomings make it difficult to work with the plan in an iterative fashion. Additionally, if she knows she is writing a Java plug in for the Eclipse environment, she knows that she does not need to consider dependencies from these projects (such as `java.*` and `org.eclipse.*`) as she knows her system already depends on these libraries. Even for this small scenario it can be difficult enough to

[1]http://logging.apache.org/log4j
[2]http://ganymede.sf.net

correctly identify all of the relevant structural dependencies without tool support, let alone tracking her decisions about each of them.

3. Task Requirements

We sought to fulfill three primary requirements while designing our dependency investigation tool. While we developed our visualization we thought of how each feature we were considering adding would support these three points. If the feature did not clearly meet these goals they were rejected.

Propagational navigation. Developers operate in a time-limited environment. Visual tool support must allow them to quickly traverse through a series of dependencies without getting lost. This is particularly important as we have previously found that developers would follow paths just to check some fact, but would have trouble getting back to their starting point [3].

Low commitment. In pragmatic reuse tasks, decisions must be made during the process of gathering knowledge about the transitive dependencies of a feature. As such, early decisions can be invalidated by knowledge discovered later. Developers should be able to easily back out of decisions, and retrace dependency paths regardless of their "current position" in the task.

Flexibility in abstraction. We cannot predict in advance how the developer will want to view any particular piece of data. As such, we needed to provide a range of alternatives from high-level abstractions to their concrete realizations in the source code. Also, we wanted the visualization to be flexible in that the developer could select any entity in the abstraction to go to its code and to add any element they were interested in into the visualization whether it was relevant to what they were viewing or not.

4 Related work

In our earlier work [3], we attempted to use a general graph visualization technique for navigating dependencies. As the systems being studied grew larger, the technique failed to scale, and industrial developers became easily disoriented.

A variety of program comprehension approaches are based on graph visualization (e.g., Rigi [5]). In attempting to be as general-purpose as possible, they tend to be ill-suited for specific tasks [8]. They also tend to ignore the particular needs of industrial developers [6] in investigating pragmatic reuse tasks and quickly deciding whether or how to pursue them. We impose a particular process model and its related navigational strategy on the developers using our approach; despite the fact that this would not be a good design choice for a general-purpose program understanding tool [9], we believe that the narrowly targeted application eliminates the need to permit a range of navigational strategies.

Likewise, a number of approaches have been developed to aid navigating source code dependencies. Of these, the most similar to our proposal is the Feature Extraction and Analysis Tool (FEAT) [7], which permits a developer to navigate from a source code entity to the entities on which it depends (fan-out) or to the entities that depend on it (fan-in). FEAT is too generic for our situation, as fan-in analysis is irrelevant; also, FEAT does not support recording decisions about dependencies, so central to our context.

A wide array of approaches have appeared in recent years for automatically or semi-automatically identifying the extent of features in source code (e.g., the work of Eisenbarth *et al.* [1]). Such approaches could be used as a starting point in a pragmatic reuse task; however, given that features of interest are often not nicely encapsulated, an intricate and inexact decision-making process would still be needed to draw the boundary between the feature and the rest of the system. The previous work on feature location does not aid in that task.

Several visual techniques exist to view graph structures as trees. Of these, TreePlus [4] is most similar to our approach. TreePlus advocates the *Plant a seed, watch it grow* metaphor for graph visualization, starting from a point and working outwards—essentially, propagational navigation. We believe that the TreePlus approach to visualization is still too generic for our purposes, and would place a greater learning burden on developers than they would be willing to shoulder because of its novel approach to presenting its information. We feel that using standard GUI widgets that developers are comfortable with is more likely to lead to adoption; however, future work will be needed to confirm or refute our conjecture.

5. Visual Approach

Our tool for creating pragmatic reuse plans is called Gilligan and is implemented as an Eclipse plug-in[3]. Gilligan's visualization involves two components: (1) three interdependent tree-list panes for abstractly representing source code dependencies (Figure 2 [top]); and (2) an editor view for displaying source code itself (Figure 2 [bottom]). To start Gilligan, the developer selects the system she wishes to investigate for reuse, as well as a target system within which she wishes to reuse the code. She then identifies at least one source code entity of interest (to her) from the source system. In Figure 2 the `startListener()` method has been selected as the starting point for investigation; this is added to the leftmost (concern) pane. Any number of nodes (which can be classes, methods, or fields) can be added to this pane. Nodes can be added or removed from this pane either via a dialog or from the other two panes (using the context menu) to the right at any time. Nodes are always shown with some form of context; that is, their package and

[3]http://eclipse.org (v3.2.1)

containing class is always visible. At any time the developer can request the source code corresponding to any node in the three panes, via selection of the appropriate context menu item.

The direct dependencies for any node that is selected in the concern view are shown in the center pane. If more than one node is selected, the union of their direct dependencies is displayed. Unlike in the concern pane, nodes only appear in the direct dependency pane as a result of the developer's selections within the concern pane. Developers can, however, hide nodes that are not of interest to them from this view.

The indirect dependencies of any selection in the direct dependency view are shown in the rightmost view. Selections in this pane do not affect the panes to the left; if a node is selected in this pane, its dependencies are added to the same pane. To differentiate these dependencies from the others in the list, the developer can click to add the node to the concern view to see both the direct and indirect dependency lists.

There are three columns in each of the tree views. The leftmost column corresponds to the element's name, and provides a descriptive icon. This icon indicates the type of the node (package, class/interface, method, or field). The icon can also be decorated to provide extra information. Dependencies on types that are solely present as binaries (i.e., class files) are annotated with a slash through their icon; if a class has sub- or supertypes these also overlay this icon, as down- or up-arrows. The second column is a coloured rectangle corresponding to the annotation the developer has placed on the node; these annotations are discussed later in this section. The third column enumerates the number of direct dependencies of a node, while the fourth enumerates the number of dependencies in the transitive closure of the node's dependencies. These two columns are useful to give the developer a quick sense of the number of dependencies down any particular path. Any node with 0 dependencies need not be investigated at all, while a node with over 100 dependencies may be crucial to investigate.

Nodes are also annotated to show if they have been visited before; nodes that have not been visited are showed in lighter text than those that have. This gives the developer a simple cue as to where she has been before.

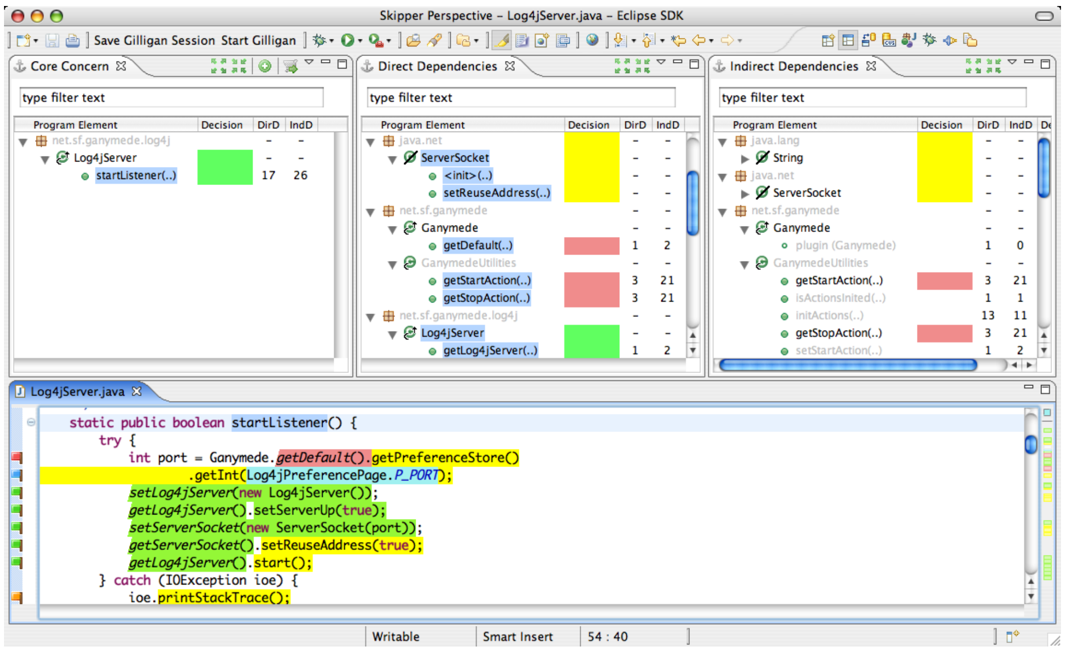
If the developer is interested in viewing a particular node in the list, she can use the text search fields at the top of each pane to filter the list and only show the nodes in which she is interested. In these cases, the nodes retain their parents so their origin can be easily seen. The tree lists can also be fully expanded or contracted as required; this can be helpful if the developer only wants a package-level overview or is particularly interested in the field- and method-level information. The lists are also fully keyboard navigable, which makes it a simple matter to quickly traverse the nodes in the list, viewing their direct and indirect dependencies.

The developer is free to examine the source code for any node. Gilligan marks up the source code (Figure 2, bottom) with the same colours as the decisions that the developer has already made.

Interaction model. The core feature of the visualization is its ability to dynamically update to show the developer the dependencies for the nodes she is investigating. The lists update based on her selections and provide information that is similar to that which she would have access through in a graph, but presented in a list-based view. Because the lists are linear and the order of their results is stable (alphabetically sorted), it is easy for the developer to return to any previous state. By allowing her to add any node to the concern (leftmost) pane, the developer has a particularly handy way of storing and retrieving nodes of interest during her investigation. This display mechanism does hide some details that would be present in a graph-based view. In particular, the origin of a dependency cannot be directly identified if multiple nodes are selected, nor can the number of times a dependency is referenced. These two facts can be derived by navigating the lists if needed, but we have informally observed that developers performing pragmatic reuse tasks are not typically interested in this information. If the developer wants to see the transitive closure of all of the structural dependencies of a node in the concern pane, all she needs to do is select all of the node's direct dependencies in the direct dependency pane. Facilitating the navigation of the dependencies highlights only half of Gilligan's utility; we also help the developer record their decisions about each dependency through various annotations.

Annotations. To convert the time spent investigating dependencies into a concrete plan, we provide an annotation mechanism on top of the interaction model. The annotations simply consist of marking nodes with different colours that the developer can use to record different decisions about the dependencies she is investigating. Green annotations correspond to dependencies on code that the developer wants to reuse. Red annotations correspond to code that she does not want to reuse. Blue annotations indicate code that performs functionality already provided within the developer's target system, but with a different interface. These three annotations are manually chosen by the developer according to her decisions as to how to triage dependencies. Yellow annotations are automatically generated by the system; these correspond to dependencies that are already provided within the target system. For example, in Figure 2 the target system includes the Eclipse and Java libraries, so all dependencies on these are automatically annotated in yellow.

The prominent nature of the annotations allows the developer to get a sense of the types of dependencies she is considering, at a glance. The developer is free to annotate dependencies while traversing them, rather than thor-



Program Element	Decision	DirD	IndD
net.sf.ganymede.log4j		-	-
Log4jServer	🟩	-	-
startListener(..)	🟩	17	26

Program Element	Decision	DirD	IndD
java.net		-	-
ServerSocket	🟨	-	-
<init>(..)	🟨	-	-
setReuseAddress(..)	🟨	-	-
net.sf.ganymede		-	-
Ganymede		-	-
getDefault(..)	🟥	1	2
GanymedeUtilities		-	-
getStartAction(..)	🟥	3	21
getStopAction(..)	🟥	3	21
net.sf.ganymede.log4j		-	-
Log4jServer	🟩	-	-
getLog4jServer(..)	🟩	1	2

Program Element	Decision	DirD	IndD	De
java.lang	🟨	-	-	
String	🟨	-	-	
java.net	🟨	-	-	
ServerSocket	🟨	-	-	
net.sf.ganymede		-	-	
Ganymede		-	-	
plugin (Ganymede)		1	0	
GanymedeUtilities		-	-	
getStartAction(..)	🟥	3	21	
isActionsInited(..)		1	1	
initActions(..)		13	11	
getStopAction(..)	🟥	3	21	
setStartAction(..)		1	2	

```
static public boolean startListener() {
    try {
        int port = Ganymede.getDefault().getPreferenceStore()
                .getInt(Log4jPreferencePage.P_PORT);
        setLog4jServer(new Log4jServer());
        getLog4jServer().setServerUp(true);
        setServerSocket(new ServerSocket(port));
        getServerSocket().setReuseAddress(true);
        getLog4jServer().start();
    } catch (IOException ioe) {
        ioe.printStackTrace();
```

Figure 2. Screenshot of Gilligan; abstract views above, annotated source code below.

oughly investigating, forgetting details, and then making decisions for unclear reasons. However, the developer is free to change her mind about the decisions at any time. While no two tasks are alike, the developer can determine whether the amount of reused code versus the other annotations makes it worthwhile for her to pursue the reuse task. Without this overall view, it can be difficult for the developer to make an informed decision about the reuse task.

6. Evaluation

To evaluate the effectiveness of our approach relative to standard IDE tools, we conducted a semi-controlled experiment. We recruited six participants, each of whom performed four tasks. Two tasks were undertaken using our approach, while the other two tasks were performed using standard IDE tools ("manually"). The participants were tasked with identifying the transitive dependencies for a fragment of source code. The number of dependencies each participant needed to identify in each task ranged from 30 to 93. Participants were given an unlimited amount of time to complete each task and were passively monitored during this time. Each participant was a software engineering graduate student, each of whom was comfortable with the Eclipse IDE and actively writes code on a regular basis.

We augmented the IDE for the manual case with the ConcernMapper plug-in[4]; the subjects used this to record those source code elements they considered for each task. Concern Mapper also served as an augmentation of the standard IDE toolset, as the subjects could use it as an index to navigate between entities they had previously added to it.

6.1 Experimental set-up

Subjects (S1 through S6) were assigned to treatment groups (G1, G2) in a randomized fashion. As we had a limited number of subjects we chose not to randomize the task (T1 through T4) order, but rather leave it fixed so the learning effect from one task to another would be the same for each subject. The subjects in G1 performed T1 and T2 with Gilligan and T3 and T4 manually, while the subjects in G2 reversed this treatment. Additionally, to decrease the learning effect, we chose each task from different systems so they would not overlap.

In addition to recording the time required to complete each task, we also recorded the nodes identified as being dependencies of the assigned fragment of interest. We compared these results to solutions we derived by carefully completing the task several times. This resulted in our knowledge the number of correct, missing, and incorrect nodes for each trial. Using these numbers we are able to report precision (ratio of relevant nodes identified to relevant and irrelevant nodes identified) and recall (ratio of relevant nodes identified to number of relevant nodes in solution) results.

Next we describe each task along with some observations made while the participants were performing them. The the size of the correct solutions for each task is shown in Table 1. Before the participants took part in the study, they

[4]http://www.cs.mcgill.ca/ martin/cm/ (v1.3.1)

Task	Classes	Fields	Methods	Total
Ganymede	4	9	17	30
HttpClient	7	22	31	60
GanttManager	12	10	28	50
Jajuk	15	47	31	93

Table 1. Numbers of dependencies in the correct solutions.

were each given a written description of what the study was about and a training task during which they could talk to the investigators as they learned about both Concern Mapper and Gilligan. The subjects had as long as they wanted to investigate the tools on the training task before starting the four experimental tasks.

Task 1: Ganymede. The first task involved investigating the dependencies for starting a Log4J Socket Server inside Ganymede, an Eclipse-based Log4J project[5]. Ganymede consists of 34 classes and 2,234 lines of code. The subjects were tasked with finding all of the transitive dependencies for `Log4JServer.startListener()` while excluding any dependences from the Java standard libraries or from the `org.eclipse.*` packages. S1, S2, and S3 used Gilligan for this task, finding all of the correct dependencies in an average of 5 minutes and 20 seconds. Performing the task manually, participants S4, S5, and S6 had a recall of 0.77 and took an average of 13 minutes 40 seconds. The recall of S4–S6 suffered because each of them failed to adequately explore the dependencies of `GanymedeUtilities.initActions()`. This one method call obscured as a condition in an `if` statement but was the root of a dependency chain containing over 18 other elements. Although they identified many of these dependencies via other paths, they ended up neglecting to identify an average of 7 structural dependencies from the solution, each of which the subjects using Gilligan found.

Task 2: HttpClient. For the second task the subjects had to identify the dependencies for parsing cookies in HttpClient[6]. This project contains 165 classes encoding 15,970 lines of code. The investigations started in `CookieSpecBase.parse(...)` and excluded any Java standard library classes. The participants performing this task manually identified significantly fewer correct dependencies than the participants supported by Gilligan (46% vs. 99%). Again, participants manually performing the task missed a large number of dependencies by failing to follow a single path through `Date.parseDate(...)`; this path led to 13 dependencies on `ParameterParser` that each of these subjects failed to detect.

[5]http://ganymede.sf.net (v0.9.3.1)
[6]http://jakarta.apache.org/httpclient (v3.0_rc1)

Task 3: GanttManager. The third task involved locating the dependencies for adding a delay to a Gantt activity in GanttManager[7]. The Gantt project comprises 555 classes and 43,247 lines of code. Each participant started in `ConstraintImpl.addDelay(...)`. Again, the participants were asked to disregard dependencies to Java standard library code. In this task, both treatment groups failed to detect a large number of the structural dependencies. Participants in the manual treatment group had an average recall of only 39% over an average of 17 minutes of investigation while the Gilligan-supported subjects had an average recall of 50% over 8 minutes and 20 seconds. While observing the participants performing this task, 5 of the 6 failed to notice that a large portion of the structural dependencies for this task (half of them) were performed by subtypes that needed to be investigated indirectly. Several of the structural dependencies led the participants to interfaces and abstract classes; only one of the participants investigated the subtypes of these interfaces. In each case only one subtype existed so it was easy to determine which type the code was actually dependent on.

Task 4: Jajuk. The final task looked at adding an item to a playlist inside the Jajuk media manager[8]. Jajuk is implemented by 227 classes comprising 30,679 lines of code. The participants were asked to ignore dependencies on Java standard libraries and on `org.apache.logging.log4j.*` packages. Each participant started their investigation in `History.addItem(...)`. This task had the greatest discrepancy between the Gilligan-supported subjects and the manual participants. The manual-treatment subjects had an average recall of 19% in 9 minutes and 40 seconds. The participants using Gilligan had an average recall of 98%, spending 11 minutes and 20 seconds to identify these dependencies. Although the manual-treatment subjects performed the task on average faster than the Gilligan-supported subjects (by 1 minute and 40 seconds), they performed significantly worse in locating each of the correct dependencies. During this task we observed the participants revisiting the same node much more than in other tasks. The manual-treatment participants in particular would visit the same method several times trying to remember if they had been there before. The manual participants each missed a branch containing 77 dependencies by not following a single path into `FileManager.getInstance()`.

6.2 Observations

Observing each of the participants while they performed the tasks yielded several insights. First, on every manual task, the participants had difficulty determining if dependencies

[7]http://ganttproject.sf.net (v2.0.4)
[8]http://jajuk.info (v1.3.0)

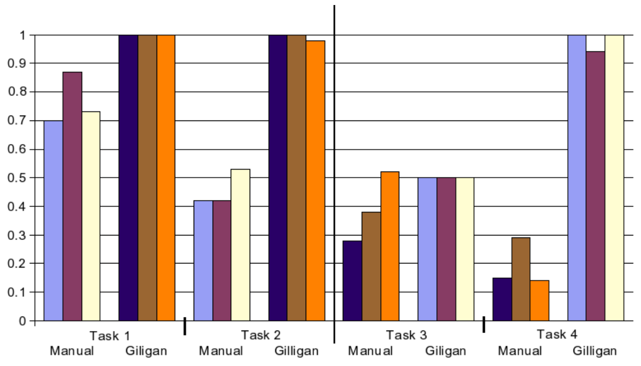


Figure 3. Recall for the four tasks.

were part of the restricted set of dependencies they did not have to consider. The participants would hover over the dependency to see its fully-qualified type or they would navigate to the dependency and scroll to the top to see its package signature. For method bodies that had a large number of dependencies that met the restrictions, the subjects often missed important dependencies as they were obscured by unimportant ones; this was the case that led to the missed path in T2. While using Gilligan, the participants would scroll though the transitive dependency list to check for any nodes that had not been annotated as a final step. While this was effective for T1, T2, and T4, it led to errors in T3 as the subjects needed to interact more with the tool to request the subtypes of the interfaces they had located.

After the participants had completed all four treatments, we performed an exit interview to see how they perceived their performance for each treatment. The participants were quite positive about their performance: on a Likert scale from 1 (very poor) to 7 (very high) they rated their confidence in their solution at an average of 6.7 for the tool-supported cases and 5 for the manual cases. The most common complaint about the manual approach was that "since you can't remember where you've been you often end up doing things over" (S4); these sentiments were echoed in similar comments by S2, S5, and S6. S1 also stated that dependencies could sometimes be "obfuscated by the source code" and that "scanning through code your brain sees chunks and if you misinterpret some part you'll just skip over [it]". This was borne out in their performance of the manual tasks.

6.3 Results

For each task, we had created an answer key that could be used to evaluate the structural elements identified by the participants. We have talked primarily in terms of recall; that is, the proportion of correct nodes identified by each subject for each task (Figure 3). From this graph we can see that participants manually carrying out the tasks generally did worse than those supported by Gilligan. Neither approach yielded high recall for T3.

As we had a correct solution we were also able to calculate precision; that is the proportion of invalid nodes to valid nodes. The manual-treatment participants achieved an average precision of 89% while the Gilligan-supported subjects achieved an average of 98%. Unlike the recall case, precision was fairly consistent across all of the tasks, so we do not present it in a graph.

Figure 4 compares the average time taken per-task against the average recall. In this chart better solutions would appear at the bottom right, while worse solutions would appear at the top left. In general from this graph we can see that the manual tasks took longer and had lower recall than the tasks for which the subjects had access Gilligan. The subjects took an average of 9 minutes per task using Gilligan and 14 minutes performing the tasks manually. Their average recall using Gilligan was 0.87 compared to 0.45 in the manual case.

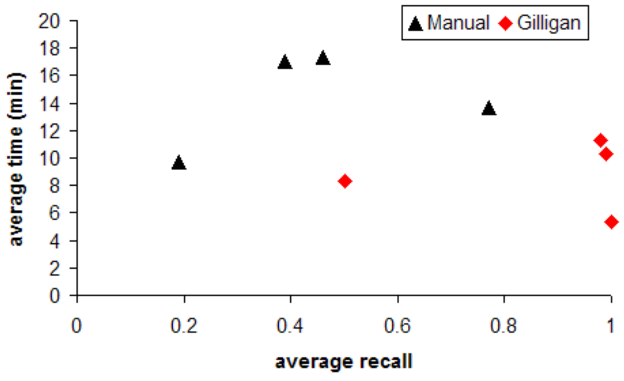


Figure 4. Average recall and time.

7. Discussion

In this section we examine issues relating to the validity of the experiment we performed, open issues surrounding our visual approach, and future directions this work may take.

7.1 Experimental validity

We conducted our evaluation to gather initial evidence into the efficacy of our approach to judge whether further investment should be made in the tool. The small sample size and the small scale of our evaluation are obvious drawbacks. While our subjects were graduate students, they were all software engineers who actively write code, and many of them have industrial experience. Furthermore, while the tasks performed by the subjects were not very large, the average recall for the manual cases was quite low; using larger tasks would likely only exacerbate this result. Our semi-controlled experiment demonstrated, for the tasks and subjects we tested, that Gilligan increased recall and decreased the time required for subjects to identify the structural dependencies compared to manual techniques. Further evaluation involving industrial developers is needed to determine the generalizability of these results. Comparing Gilligan to other approaches such as TreePlus could also be beneficial.

7.2 Open issues

Gilligan performed well during the evaluation. However, in the third task the developers failed to locate many of the relevant dependencies. This shortcoming occurred because, by default, the tool is designed to show dependencies from method calls and field references. In this task, many of the dependencies originated in subtypes in the inheritance hierarchy. While visual cues for this information were provided by the tool, we believe that they were not effective because the developers trusted its default output. Gilligan will have to be modified to provide more inheritance-aware information by default in the future.

Is this a software visualization approach? Gilligan is a hybrid approach combining visualization techniques with standard GUI widgets. We believe Gilligan is indeed a visualization approach because it abstracts the data it presents from its model, and it relies on the graphical cues given for the annotations. For instance, during the study subjects would check the completeness of their solutions in the Gilligan-supported cases by selecting all of the direct dependencies for the fragment they were to investigate. They would then quickly scroll through the resulting list of the transitive dependencies to ensure that each of them was annotated with a colour. The subjects would also open the source code view for nodes they were interested in to quickly determine the coverage of the annotation colours in that source code. By using standard GUI widgets, this hybrid approach is designed to be easy for developers to learn to interact with.

In Section 3 we identified three traits that we designed Gilligan to support. In terms of *propagational navigation*, Gilligan quickly shows developers the dependencies of any node in the system. They do not have to compose specific queries; the dependencies are automatically displayed based upon their selection. As the navigation is based on simple selections in lists, the quick investigation of dependency paths, and back-tracking out of them to previous states without becoming lost, is supported. This supports *low commitment*. By providing both the abstract list-based views along with the annotated source code the developer has the ability to view the system at varying degrees of detail. This *flexibility of abstraction* allows the developer to investigate each part of the system at the level of fidelity that best meets their needs. By supporting these three traits, Gilligan is able to help developers quickly identify the information they need to perform their reuse planning tasks.

We created Gilligan to help developers navigate and annotate structural dependencies for the purposes of creating pragmatic reuse plans. However, other applications for this type of approach exist. Currently developers perform refactoring tasks one step at a time. By creating a refactoring plan, developers could select multiple different refactorings and apply them all at once. This could allow for more in-depth refactoring tasks. Our investigative approach could also be used as a front-end for impact analysis tasks. Developers could quickly identify those pieces of code that their code relies on before carrying out a task.

8. Conclusion

We have presented an approach for visualizing source code dependencies for the purpose of creating reuse plans to be used while evaluating pragmatic reuse tasks. This approach relies on a co-located list metaphor for visualizing direct and indirect dependencies. Annotations can be added to any node in the visualization to allow the developer to tag nodes with different decisions they have made. Once the developer has triaged all of the dependencies in the system they can use the resultant reuse plan to make an informed decision about the feasibility of pursuing the reuse task. We have evaluated our visualization through a small-scale, semi-controlled experiment and found it to be more effective (in terms of recall and time savings) at providing the developer with the correct dependencies than when determined manually.

References

[1] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[2] I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Trans. Vis. Comp. Graph.*, 6(1):24–43, 2000.

[3] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proc. Int'l Conf. Softw. Eng.*, 2007. Pre-press version available: http://lsmr.cs.ucalgary.ca/pubs.

[4] B. Lee et al. TreePlus: Interactive exploration of networks with enhanced tree layouts. *IEEE Trans. Vis. Comp. Graph.*, 12(6):1414–1426, 2006.

[5] H. A. Müller and K. Klashinsky. Rigi: A system for programming-in-the-large. In *Proc. Int'l Conf. Softw. Eng.*, pages 80–87, 1988.

[6] S. P. Reiss. The paradox of software visualization. In *Proc. Int'l Wkshp. Visualizing Softw. for Underst. and Analysis*, pages 59–63, 2005.

[7] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. Int'l Conf. Softw. Eng.*, pages 406–416, 2002.

[8] T. Schäfer and M. Mezini. Towards more flexibility in software visualization tools. In *Proc. Int'l Wkshp. Visualizing Softw. for Underst. and Analysis*, pages 64–69, 2005.

[9] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Sci. Comp. Progr.*, 36(2-3):183–207, 2000.