Semi-Automating Pragmatic Reuse Tasks

Reid Holmes and Robert J. Walker
Laboratory for Software Modification Research
University of Calgary
Calgary, Alberta, Canada
rtholmes, rwalker@cpsc.ucalgary.ca

Abstract

Developers undertaking a pragmatic reuse task must collect and reason about information that is spread throughout the source code of a system before they can understand the scope of their task. Once they have this understanding, they can undertake the manual process of actually reusing the source code within their system. We have created a tool environment to help developers plan and perform pragmatic reuse tasks enabling them to reason about and perform larger reuse tasks than they generally feel comfortable attempting otherwise.

1. Overview

Developers commonly encounter situations where they need functionality that they know exists in an existing project. Rather than re-develop that functionality, or componentize it (with the potential to break the other project), a *pragmatic reuse* approach is sometimes preferable [1, 5, 2]. Pragmatic reuse tasks involve extracting functionality from an existing system and reusing it within another system. As such functionality is not necessarily designed for the needed reuse scenario, the developer must determine *where* to define the boundary between the code to be reused and that not to be reused, and *how* to cope with the dependencies that will dangle across this boundary after the functionality is carved out.

We have developed a reuse environment to aid developers in pragmatic reuse tasks. Our environment, called G&P, helps developers both to plan and to perform pragmatic reuse tasks. G&P currently consists of two components: the planning component and the enactment component. We describe each in turn.

The planning component (called Gilligan) allows the developer to simultaneously investigate the dependencies from their desired functionality, and to construct a lightweight plan about how these should be handled [2]. While these structural dependencies are crucial to understanding pragmatic reuse tasks, it is burdensome to discover them by inspecting the source code. While the direct dependencies of any piece of code are clear to see, it is difficult to see how many indirect dependencies any one dependency requires without navigating through many source files. For this reason, Gilligan provides developers with a list-based abstraction of the dependencies within the code that the developer wants to reuse [3]. This abstraction supports quick navigation of the statically-derivable structural dependencies within the system, enabling the developer to investigate each dependency of their desired functionality without having to read through many source files. As the developer investigates the code they can annotate the abstraction recording their decisions about any dependency: *Do I want to reuse this? Should I remap this to existing functionality within my system? Is this code common between the source system and my project?* Ultimately, this annotated view encodes a plan of the reuse task; this plan describes which elements should be reused and how any dangling dependencies should be managed. The plan can be automatically evaluated for completeness, and the developer's attention drawn to any remaining, dangling references.

Small decisions made by the developer while planning out their reuse task can have large consequences on the amount of work they must perform. For instance, if the originating system used a logging infrastructure different from the one that the developer's system uses, they may have to either remove all references to the logging framework or change each one to instead log compatibly with their system. While planning a pragmatic reuse task, the developer is constantly weighing the cost of reusing a specific dependency against the cost of having to replace or eliminate it once the code is reused in their system.

The enactment component (called Procrustes) automates the enactment of the reuse plan [4]. Using the plan, Procrustes automatically extracts the relevant code from its originating system, transforms the source code as necessary to minimize the number of compilation errors arising from

removing the code from its originating system, and injects it into the developer's system. By resolving many of the compilation errors the developer would have to otherwise manually fix, Gilligan helps shift their focus from low-level trivial compilation problems to more high-level conceptual issues that may interfere with their reuse task. Automation also enables the developer to quickly iterate on their reuse plan and see how their decisions are reflected in the source code. This enables developers to apply their skills in working with source code, while still enabling them to quickly investigate alternative plans using the abstract representation of the code's dependencies. By supporting quick iteration, Gilligan allows the developers to investigate alternatives in the reuse plan, which would otherwise be overwhelming to perform manually.

We have performed both quantitative and qualitative studies of the G&P environment and have found that developers can use it to effectively plan and perform reuse tasks.

2. Scenario

In this scenario we will briefly describe how the developer builds, enacts, and iterates on a pragmatic reuse task. The aTunes[1] music system has a panel that displays related artists for any song that it is currently playing. A developer decides they want to reuse this feature within their own system. This entails finding the relevant parts of aTunes, extracting them, and integrating the code into their system.

The developer starts by using Gilligan and selecting `AudioScrobblerService.getSimilarArtists()` as the initial point of investigation. Gilligan points out that this method has 15 direct and 103 indirect dependencies. The developer decides that this code is relevant to their reuse task and annotates it for reuse (as shown by the green bar in Figure 1). The developer then presses the enact button as they want to see, in the code, how the reuse task is shaping up. At this point they are faced with 106 errors that they must resolve. Continuing their investigation, the developer finds that reusing `AudioScrobblerCache` increases the number of errors they must resolve by 79; because of this they decide to investigate this dependency in depth as the cost of reusing it seems very high. Upon further investigation this class simply stores a local copy of related artists and does not actually contribute directly to their location; thus, this dependency is rejected.

The final reuse plan for this task involved reusing 424 lines of code from 9 classes in less than 20 minutes. Gilligan is able to resolve 63 compilation errors automatically leaving the developer with only 3 errors they have to manually resolve. Visually inspecting the code the developer finds they can be easily resolved and chooses to fix them manually rather than go back to the abstract view of the system.

[1] http://atunes.sf.net/

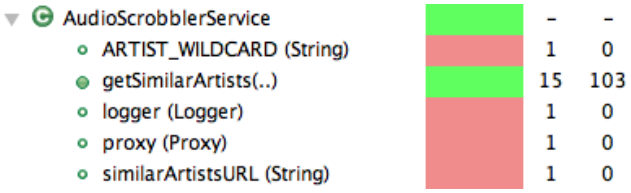

Figure 1. Snippet of a reuse plan.

3. Related Work

While the majority of reuse research has focused on black box strategies, the importance of pragmatic approaches has been previously identified [6, 1, 5]. We have previously described how pragmatic reuse plans work [2]. The details concerning how Gilligan automates the enactment of reuse plans have also been described [4].

4. Conclusion

Our tool support helps developers plan and perform pragmatic reuse tasks in three ways. First, it helps them investigate the costs of reusing any particular piece of code. Secondly, it enables them to investigate alternative reuse strategies. Finally, it relieves a large percentage of manual work the developer must engage in to perform the reuse task by automatically copying and modifying the source code for them. By providing these benefits we have found developers are more comfortable attempting larger reuse tasks than they would otherwise try manually.

References

[1] J. R. Cordy. Comprehending reality – practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the International Workshop on Program Comprehension*, pages 196–207, 2003.

[2] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proceedings of the International Conference on Software Engineering*, pages 447–457, 2007.

[3] R. Holmes and R. J. Walker. Task-specific source code dependency investigation. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, pages 100–108, 2007.

[4] R. Holmes and R. J. Walker. Lightweight, semi-automated enactment of pragmatic-reuse plans. In *Proceedings of the International Conference on Software Reuse*, 2008.

[5] C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *Proceedings of the Working Conference on Reverse Engineering*, pages 19–28, 2006.

[6] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.