Reuse for Code-Level Generalization

Alison Teoh
Department of Computer Science
University of Maryland,
College Park, MD 20742, USA
alison@cs.umd.edu

ABSTRACT

This paper describes how code reuse combined with good programming practices can also be used as a means to attain code-level generalization of certain programs. It presents some basic principles to be used when refactoring code for reuse and presents a case study in which the functionality of two AI planners are easily combined via application of these principles.

1. INTRODUCTION

The concept of code reuse has been around for decades, originally viewed as a "silver bullet" for economical software production. Reusing software assets could save time and money as programmers would not have to waste working hours rewriting code that had previously been written. Over the years, however, controversy has arisen regarding whether or not the benefits outweight the drawbacks and difficulties of practically incorporating code reuse in industry. Similarly, there has been great debate over what should be considered the best approaches to code reuse, and how to measure the effectiveness of such practices.

Despite the controversies regarding code reuse, it is indisputable that many software systems bear similar properties and employ many of the same theoretical constructs within the underlying mechanics. In this work, we describe some basic design principles to guide the restructuring of Lisp code using the Common Lisp Object System (CLOS) for effective reuse and attaining code-level generalization. We focus on the well-known AI planning system SHOP2 [12] and describe our work on refactoring the original SHOP2 implementation in order to emphasize code reuse in the light of our design principles.

We then describe a case study in which we employed these principles to easily generalize the functionality of the planner SHOP2 into that of another planner ND-SHOP2 and effectively incorporate support for planning in nondeterministic domains. The generalization from SHOP2 to ND-SHOP2 has been theoretically demonstrated before in [7], along with a broader class of AI planners in deterministic domains that can be generalized to work effectively in nondeterministic domains. Previously, this transformation involved the guidelines for the theoretical generalization of the deterministic planner, which a programmer would then use to implement the new, nondeterministic planner. However, by employing a few basic principles of code refactoring, it is possible to attain *code-level generalization*, in which the same code is used to carry out the functionality of both planners.

2. RELATED WORK

Previous works describing code reuse have focused primarily on aspects of business efficiency. Studies of the quality of various metrics abound [2, 8, 13] as experts debate the supposed benefits of reusing code. Factors that play into that debate are the overhead of writing reusable code and the issues associated with retrieving it later [9, 17]. With the advent of object-oriented programming, the concept of code reuse moved beyond simple copy-and-paste techniques and into more generalized, systematic processes. The benefits and drawbacks of code reuse "in the small" versus large-scale endeavors have been well-debated [4, 5, 6]. Similarly, there has been a great deal of contention regarding the best approach to code reuse, most of which has focused on modifications to classical object-oriented programming languages [14, 18, 16]. To the best of our knowledge, there has been little research related specifically to the reuse of Lisp code.

3. PRINCIPLES OF REUSE

In order to refactor pre-written code, we consider the process in two basic steps: "Modularization" and "Code-Level Generalization," which illustrate the concepts of functional decomposition and knowledge engineering, respectively.

3.1 Modularization

Good programming practices always call for some level of modularization [15]. While object-oriented languages inherently enforce this somewhat through the concept of encapsulation, functional programming often seems to breed large, dense, one-file programs. Since functions are no longer tied to specific classes, programmers using these languages often lose sight of the importance that modularization bears on readability and potential for reuse.

Large programs can often be broken into several subsystems, each of which has potential to be reused in other projects. These subsystems ought to be separated and arranged in a sensical directory-style structure. Functions and classes of similar nature and function should likewise be grouped together in the same file to maximize the probability that it might be reused in entirety, as well as to enable quick location of various functions and classes. There are many packaging and system definition facilities freely available that enable this modularization when programming in Lisp.

3.2 Code-level Generalization

3.2.1 Language Matters

The Common Lisp Object System (CLOS) [1] incorporates the object-oriented paradigm into the Common Lisp framework. CLOS allows users to define class objects and hierarchies of those objects, much like in any other object-oriented system. A key difference, however, is that CLOS allows for multiple inheritance, meaning an object can simultaneously be an instance of two different classes.

Another aspect of CLOS that sets it apart from other object-oriented systems is that methods are not assigned to classes, operating on message-passing. Instead, generic functions are declared, and methods associated with the generic function dictate class-specific operations to be performed. When a method with a given name is called, all methods associated with the generic function are examined; a dispatcher selects the most relevant choice based on the classes of the objects specified in the parameter list.

In addition to multiple dispatch, another unique feature of CLOS's generic method system is the ability to combine and chain methods. The interactions of various method are specified by the method roles assigned: a defined method may specify that it perform some function before, after, or around the primary method in order to carry out implementation of the generic method. This feature allows responsibilities in overall method functionality to be split between a class and its superclass as the method defined in each takes on a different role.

3.2.2 CLOSification

In order to attain code-level generalization in the way we intend, code implemented in the Common Lisp framework must be refactored so as to make use of the properties of the CLOS. This refactoring process is affectionately referred to as "CLOSification" and primarily involves redefining structs as classes and defining generic functions for those methods that require dispatch. One of the key factors in this step is identifying which objects control the method that should be chosen when a function is called; it is unlikely that the dispatcher need examine the entire list of method parameters in order to choose that method which is best-suited to a given situation.

4. CASE STUDY - SHOP2 & ND-SHOP2

4.1 The Planners

We began with the implementation of a forward-chaining deterministic planner, SHOP2 [12], which was implemented in Common Lisp. In a work published in 2004 [7], Kutur and Nau claimed that nondeterministic forward-chaining planners were simply generalizations of their previously-established deterministic counterparts. They then presented a technique for generating the nondeterministic planner ND-SHOP2 from the framework of SHOP2. The basis for their process of nondeterminizing planners is based on an abstraction of forward-chaining planning procedure, FCP. FCP and the corresponding nondeterministic version, ND-FCP are shown in Figure 1[7].

SHOP2 is an instance of the FCP procedure as follows: the planning engine subsystem of SHOP2 itself performs a simple forward-chaining procedure as that represented in FCP. In SHOP2, a problem consists of an initial state, a task network which encodes the goals of the planning problem, and a domain description. This domain description contains user-defined methods that can decompose the tasks in the input network into smaller subtasks. The α action-generation function that decreases the relevant search space in FCP is implemented in SHOP2 through use of these methods. More specifically, given a state s and a task network w, the planner applies its methods to w to determine which actions are applicable in state s. Applying these actions modifies w. SHOP2 nondeterministically chooses one of those actions, and applies it in s to generate a successor state. It then continues with the current task network until a solution is found – i.e., until there are no tasks left to be accomplished in the world. ND-SHOP2 operates in much the same way, except that applying an action in state s produces a nondeterministic result. Thus, application of any action generates a set of possible successor states.

In this case study, we outline the restructuring of the SHOP2 code in accordance with the principles outlined in Section 3 and then demonstrate how we were easily able to incorporate the nondeterministic functionality of ND-SHOP2 into the code.

4.2 SHOP2 Modularization

SHOP2 was originally implemented entirely within one file, several thousand lines long. Some attempts at modularization had previously been made, but the organization was not thorough or systematic. Our first step was to restructure the code in a sensical way. Once the code had been split into various files, we packaged the files using Another System Definition Facility (ASDF), a freely available piece of software for use with Common Lisp. Figure 2 illustrates the reorganization and modularization of the SHOP2 code.

We began by ensuring that all functions and classes within a file were related and that the file was named sensically. For example, all functions that had to do with parsing the domain and problem specified by the user were moved to single file named "input.lisp." By following through with this process, all SHOP2 code was disseminated between a couple dozen files.

The planner had previously been conceptualized as a combination of subsystems, so the next step was simply a matter of organizing the files into folders that accurately reflected their functions in the planner. All files with functions having to do with input parsing, printing plans, etc were moved to a folder that represented the I/O system. Files containing functions that perform the actual plan generation were clustered together in the "planning-engine" folder. We continued grouping files in this manner until everything but the top-level files were organized in subdirectories.

4.3 SHOP2 CLOSification

As mentioned earlier in Section 3, the first step was redefining structs as classes. We then identified the functions over which we would like to dispatch methods and defined a generic function. Methods were then implemented to dispatch over the desired object. It was at this point that we started adding support for functionality that we would like to see in the future. Figure 4 shows the definition of a generic

Procedure FCP(s_0, g, O, α);
 $\pi \leftarrow \emptyset$; $s \leftarrow s_0$
 loop
  if s satisfies g then return(π)
  $A \leftarrow \{(s,a) \mid a$ is a ground instance of an operator
   in O, a is applicable to s, and $a \in \alpha(s)\}$
  if $A = \emptyset$ then return($failure$)
  nondeterministically choose $(s,a) \in A$
  $\pi \leftarrow \pi \cup \{(s,a)\}$
  $s \leftarrow \gamma(s,a)$

Procedure ND-FCP(S_0, g, O', α')
 $\pi \leftarrow \emptyset$; $S \leftarrow S_0$; $solved \leftarrow \emptyset$
 loop
  if $S = \emptyset$ then return(π)
  select a state $s \in S$ and remove it from S
  if s satisfies g then insert s into $solved$
  else if $s \notin S_\pi$ then
   $A \leftarrow \{(s,a) \mid a$ is a ground instance of an operator in O', a is applicable to s, and $a \in \alpha'(s)\}$
   if $A = \emptyset$ then return($failure$)
   nondeterministically choose $(s,a) \in A$
   $\pi \leftarrow \pi \cup \{(s,a)\}$
   $S \leftarrow S \cup \gamma(s,a)$
  else if s has no π-descendants in $(S \cup solved) \setminus S_\pi$
   then return($failure$)

Figure 1: The abstract planning procedure FCP. The deterministic version appears on the left and the nondeterministic version appears on the right. Note that the underlined lines in the nondeterministic version correspond with the pseudocode in the nondeterministic FCP description.

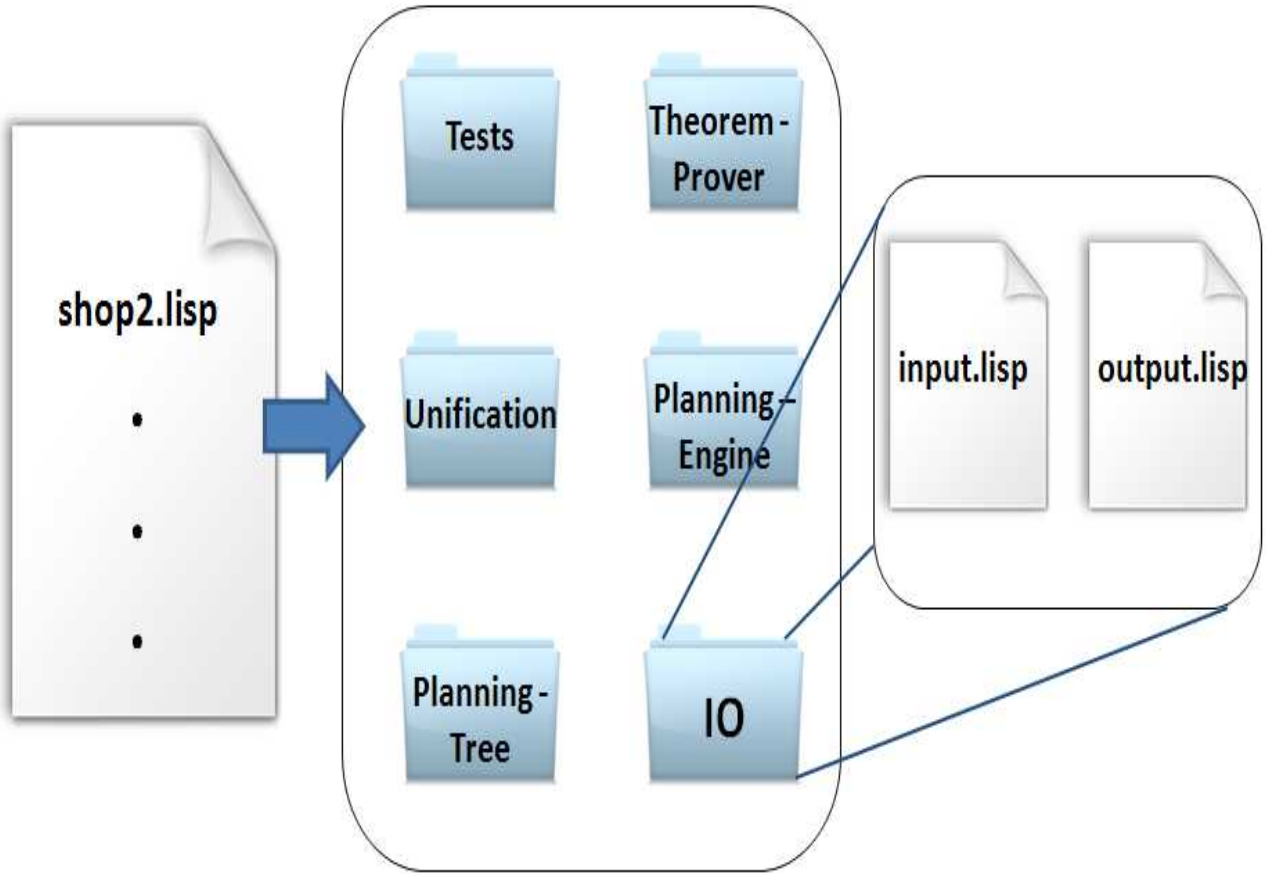

Figure 2: This is an illustration of the reorganization applied to the SHOP2 code. The code began as a single file and was split into subsystems.

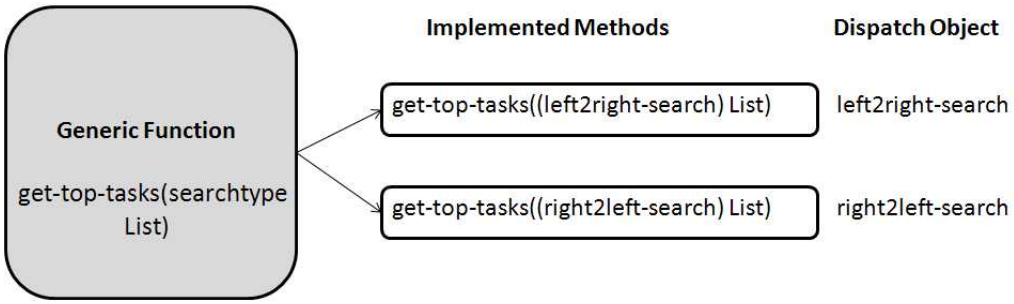

Figure 3: This is an illustration of how method dispatch works in CLOS.

```
(defgeneric get-top-tasks (searchtype List)
(:documentation "returns the list of top-tasks in the Main
List according to searchtype"))
%
%
(defmethod get-top-tasks ((searchtype left2right-search) L)
This method implements the generic get-top-tasks function
by returning the left-most "top tasks")
%
%
(defmethod get-top-tasks ((searchtype left2right-search) L)
This method implements the generic get-top-tasks function
by returning the right-most "top tasks")
```

Figure 4: Definition of a generic function and two methods which implement it. Methods are dispatched on the type of the searchtype object.

function and then two methods which implement that function. Figure 3 illustrates the knowledge flow of the method dispatching process.

For example, SHOP2 performs planning over hierarchical task networks (HTNs) and automatically assumes that task decomposition will take place in a left-to-right manner. A function called "get-top-tasks" is responsible for examining the list of remaining tasks in the network and returning the tasks furthest to the left that have yet to be decomposed into subtasks. However, we have reason to believe that it may be interesting to research the performance of a right-to-left decomposition; this choice would be made at the user's discretion, specified by creation of an object of type Searchtype. We implemented a searchtype class which then became superclass to two object classes, "left2right-search" and "right2left-search." We were then able to specify method dispatch over the "get-top-tasks" function, according to the type of Searchtype object passed to the function call.

4.4 Applying Functionality from ND-SHOP2

After refactoring the SHOP2 code, we were ready and able to simply add methods that dispatch over the domain and incorporate nondeterministic capabilities into the SHOP2 planner. As noted previously, the general planner procedure remains the same as in a deterministic setting with the small change that lists of states need be considered rather than a single state. The primary changes came in the form of recognizing the different type of domain, and then dispatching as appropriate.

In SHOP2 there is a group of methods that perform various aspects of the search function in planning. The search is broken into three cases depending on the type of the next task to be completed: null, primitive, and nonprimitive. Each case is taken care of with its own "seek-plans" function. During the CLOSification phase of the SHOP2 refactoring, these methods were written so as to dispatch over the domain of the current problem. In this way, once the new domains are recognized as being nondeterministic, the function dispatcher merely directs the planner to the method that had been implemented specifically to deal with that domain. Since the ND-SHOP2 planner was already written in full, we did not have to implement the nondeterministic methods from scratch, but simply add them to the appropriate SHOP2 files. (In the case where new functionality is added to a project but not pre-written elsewhere, it is simply a matter of implementing the new methods.) Other methods perform identically for planning in either type of domain, so we were able to reuse these methods without any modification whatsoever.

5. CONCLUSION

Code reuse has been a point of interest for decades. The goal of this paper was to introduce and demonstrate a different aspect of code reuse than that which is generally considered. In this paper we presented some basic principles for refactoring Common Lisp code such that it might be generalized at the code-level and gain added functionality with minimal adjustment. We discussed the importance of modularization for maximum readability and reuse. We also looked into the object-oriented concepts made possible in Common Lisp through CLOS and discussed how these features can be taken advantage of in order to attain potential for code-level generalization. Finally, we presented a case study in which we took two planners which had previously been shown to be theoretical generalizations of each other and easily incorporated the functionality of the second into the actual code of the first. This added functionality actually generalizes the capabilities of the planner without adding a lot of additional code, demonstrating that good programming practices in conjunction with code reuse can indeed result in code-level generalization.

6. REFERENCES

[1] L. G. DeMichiel and R. P. Gabriel. The common lisp object system: an overview. In *European conference*

on object-oriented programming on ECOOP '87, pages 151–170, London, UK, 1987. Springer-Verlag.

[2] P. Devanbu, S. Karstu, W. Melo, and W. Thomas. Analytical and empirical evaluation of software reuse metrics. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 189–199, Washington, DC, USA, 1996. IEEE Computer Society.

[3] A. Dozsa, T. Gîrba, and R. Marinescu. How lisp systems look different. In *CSMR*, pages 223–232. IEEE, 2008.

[4] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[5] R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002.

[6] R. E. Grinter. From local to global coordination: lessons from software reuse. In *GROUP '01: Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, pages 144–153, New York, NY, USA, 2001. ACM.

[7] U. Kuter and D. Nau. Forward-chaining planning in nondeterministic domains. In *AAAI-2004*, 2004.

[8] W. C. Lim. Why the reuse percent metric should never be used alone, 1999.

[9] J. Long. Software reuse antipatterns. *SIGSOFT Softw. Eng. Notes*, 26(4):68–76, 2001.

[10] A. Mockus. Large-scale code reuse in open source software. In *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, page 7, Washington, DC, USA, 2007. IEEE Computer Society.

[11] D. Nau, T.-C. Au, O. Ilghami, U. Kuter, H. Munoz-Avila, J. W. Murdock, D. Wu, and F. Yaman. Applications of shop and shop2. *IEEE Intelligent Systems*, 20(2):34–41, 2005.

[12] D. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. Shop2: An htn planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[13] D. L. Nazareth and M. A. Rothenberger. Assessing the cost-effectiveness of software reuse: a model for planned reuse. *J. Syst. Softw.*, 73(2):245–255, 2004.

[14] G. Neumann and U. Zdun. Enhancing object-based system composition through per-object mixins. In *APSEC '99: Proceedings of the Sixth Asia Pacific Software Engineering Conference*, page 522, Washington, DC, USA, 1999. IEEE Computer Society.

[15] P. Norvig. Tutorial on good lisp programming style. In *Proceedings of the Lisp Users and Vendors Conference*, 1993.

[16] J. Palsberg and M. I. Schwartzbach. What is type-safe code reuse? In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 325–341, London, UK, 1991. Springer-Verlag.

[17] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 905–908, New York, NY, USA, 2006. ACM.

[18] N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 206, Washington, DC, USA, 1998. IEEE Computer Society.