Fine-Grained Concept Indexing of Java Programming Contents Using JavaParser

Submitted in fulfillment of the requirement of the
ISSP 2990 Independent Study course

By:

Roya Hosseini

Advised By:

Peter Brusilovsky

Intelligent Systems Program
University of Pittsburgh
Pittsburgh, USA

Fall 2013

Acknowledgments

I would like to thank my advisor, Dr. Peter Brusilovsky for the valuable advice and support he has given me during my independent study for fall 2013.

Abstract

The multi-concept nature of problems in the programming language domain requires fine-grained indexing which is critical for sequencing purposes. In this paper, we propose an approach for extracting this set of concepts in a reliable automated way using the JavaParser tool. To demonstrate the importance of fine-grained sequencing, we provide an example of how this information can be used for problem sequencing during exam preparation.

Contents

1 Introduction

One of the oldest functions of adaptive educational systems is guiding students to the most appropriate educational problems at any time during their learning process. In classic ICAI and ITS systems, this function was known as task sequencing [1; 6]. In modern hypermedia-based systems, it is more often referred to as navigation support. The intelligent decision mechanism behind these approaches is typically based on a domain model that deconstructs the domain into a set of knowledge units. This domain model serves as the basis of a student overlay model and as a dictionary to index educational problems or tasks. Considering the learning goal and the current state of student knowledge reflected by the student model, various sequencing approaches are able to determine which task is currently the most appropriate.

An important aspect of this decision process is the granularity of the domain model and the task indexing. In general, the sequencing algorithm can better determine the appropriate task if the granularity of the domain model and the task indexing is finer. However, fine-grained domain models that dissect a domain into dozens or hundreds of knowledge units are much harder to develop and to use for indexing. As a result, many adaptive educational systems use relatively coarse-grained models where a knowledge unit corresponds to a sizable topic of learning material, sometimes even a whole lecture. With these coarse-grained models, each task is usually indexed with only 1-3 topics. In particular, this approach is used by the majority of adaptive systems in the area of programming [2; 4; 5; 7].

Our prior experience with adaptive hypermedia systems for programming [2; 4] demonstrated that adaptive navigation support based on coarse-grained problem indexing is a surprisingly effective way to guide students through their coursework, yet it doesn’t work well in specific cases such as remediation or exam preparation. In these special situations, students might have a reasonable overall understanding of the content (i.e., coarse-grained student model registers good level of knowledge), while still suffering some knowledge gaps and misconceptions that could only be registered using a finer-grained student model. In this situation, only a fine-grained indexing and sequencing tool is able to suggest learning tasks that can address these gaps and misconceptions.

To demonstrate the importance of fine-grained indexing, we look to a system called Knowledge Zoom (KZ). The goal of KZ is to help the students identify their course knowledge gaps and provide tools to bridge these gaps in an effective way. The first part of this dual goal is supported by the KE component, a concept-based hierarchical zoomable open student model.

The second goal is supported by the KM, a concept-based adaptive problem sequencing tool. The interface of KZ (Fig.1) provides direct access to the Knowledge Explorer (KE) model and a button to launch the Knowledge Maximizer (KM). KZ is based on a concept-level model of knowledge about Java and OOP. This model is formed by a subset of concepts from the Java ontology-http://www.sis.pitt.edu/~paws/ont/java.owl built by the PAWs lab. The Java ontology includes 344 concepts organized into an 8-level tree. The learning content in KZ is formed by 103 parameterized self-assessment questions that were developed in our team as a part of an earlier project [4]. Each question is indexed with ontology concepts. The indexing classifies the prerequisite concepts that should be known before approaching the question and the outcome concepts to be mastered by working with the question. The number of concepts associated with a single question ranges from 5 to 52 (0 to 41 prerequisites, 1 to 12 outcomes). These questions cover the 188 most important concepts of Java which form the KZ domain model.

The Knowledge Explorer (KE)

KE is a multi-level open student model visualized with a zoomable Treemap. The information presented by KE is an overlay model of Java Knowledge based on the KZ ontological domain model. The overlay student model in KZ is maintained by a user modeling service, PERSEUS [8], which updates the model after every attempt to answer a question and changes the knowledge level of concepts related to the question.

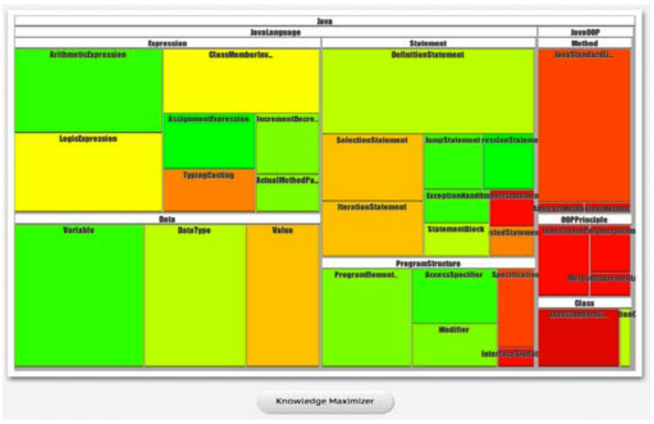


Fig. 1. The KnowledgeZoom interface showing the top level of the Knowledge Explorer map and a button to launch Knowledge Maximizer.

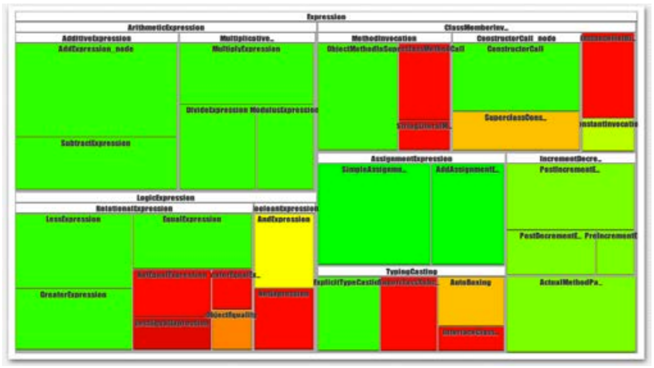


Fig. 2. Zooming on the node *Expressions* (top left corner in Fig. 1) reveals next level of the concept hierarchy. Now the user can see that the node LogicExpression that has intermediate knowledge as a whole (shown as yellow) consists of several well learned and several unknown concepts.

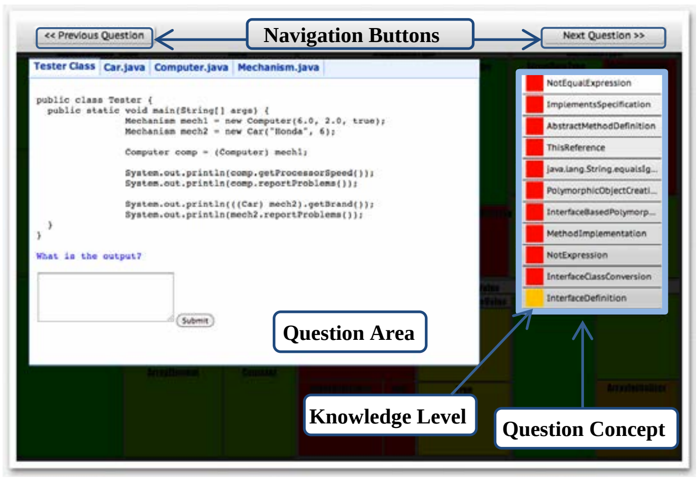
A zoomable Treemap was selected to present the student model due to its relatively large size and hierarchical nature. The Treemap layout shows only four levels of concept hierarchy starting from the current top node and hiding lower-level nodes behind its ancestor node. The user, however, can zoom in any node. After zooming in, the node expands becoming the top node and occupying the whole view. Zooming-in immediately exposes previously hidden levels of hierarchy. For example, Fig. 2 shows the results of zooming into a second level concept, Expression shown in the top left quadrant of Fig. 1.

In the Treemap layout, each node (a concept in the Java ontology) is shown as a colored rectangle. A leaf concept of the ontology corresponds to a terminal node of the Treemap. The *size* of a node represents the importance of a concept in the context of Java language and its chance to be checked as part of the exam. We measure it by counting how many questions are related to the leaf concept corresponding to this leaf node in the Treemap. Since the number of exercises related to nodes can be quite different, which leads to a large difference in the node sizes, we use the $log_2(size)$ to moderate the differences. The *color* of a node represents the level of concept knowledge demonstrated by a student. We use 10 colors from red to green to represent the progression from weaker to stronger knowledge.

In a hierarchical zoomable layout, a leaf node directly represents the importance and knowledge level of a concept with its size and color respectively, while each intermediate node accumulatively aggregates importance and concept knowledge from its child nodes. As a result of the aggregation, the upper-level views show overviews of students' state of knowledge on higher levels (Fig. 1), while being able to explore detailed knowledge of every concept as zooming into lower levels of the ontology (Fig. 2). The calculation of the aggregated size and color is important to bridge the gaps between lower and higher levels of views. In KE, the size aggregation is provided by Treemap. For the color aggregation, the color of an intermediate node is the average color of its direct child nodes weighted with their sizes in order to reflect the importance of the associated concepts.

The Knowledge Maximizer (KM)

Knowledge Maximizer [3] that uses fine-grained concept-level problem indexing to identify gaps in user knowledge for exam preparation. This system assumes a student already completed a considerable amount of work: thus, the goal is to help her define gaps in knowledge and try to redress them as soon as possible. Fig. 3 represents the Knowledge Maximizer interface. The question with the highest rank is shown first. The user can navigate the ranked list of questions using navigation buttons at the top. The right-hand side of the panel shows the list of fine-grained concepts covered by the question. The color next to each concept visualizes the student's current knowledge level (from red to green). Evaluation results confirm that using fine-grained indexing in Knowledge Maximizer has a positive effect on students' performance and also shortens the time for exam preparation.



```
public class Tester {
  public static void main(String[] args) {
                Mechanism mech1 = new Computer(6.0, 2.0, true);
                Mechanism mech2 = new Car("Honda", 6);

                Computer comp = (Computer) mech1;

                System.out.println(comp.getProcessorSpeed());
                System.out.println(comp.reportProblems());

                System.out.println(((Car) mech2).getBrand());
                System.out.println(mech2.reportProblems());
  }
}
```

What is the output?

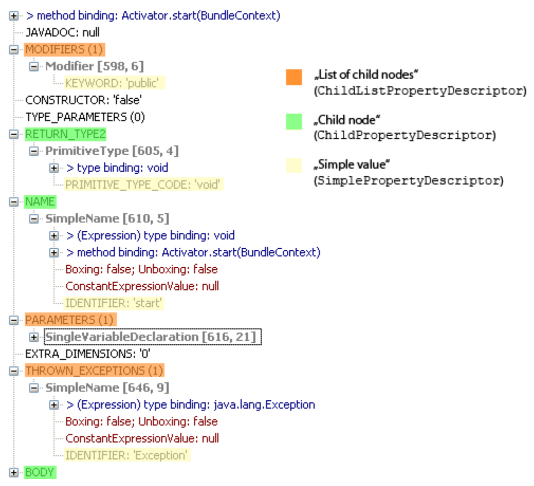
Fig. 3. The Knowledge Maximizer interface.

The problem with finer-grained indexing, such as that used by the Knowledge Maximizer, is the high cost of indexing. While a fine-grained domain model has to be developed just once, the indexing process has to be repeated for any new question. Given that most complex questions in our domain involve more than 50 concepts each, the high cost of indexing effectively prevents an increase in the number of problems represented in the system. To resolve this problem, we developed an automatic approach to fine-grained indexing for programming problems in Java based on program parsing. This approach is presented in the following section.

2 Java Parser

Java parser is a tool that we developed to index Java programs according to concepts in a Java ontology developed by our group (http://www.sis.pitt.edu/~paws/ont/java.owl). This tool provides the user with semi-automated indexing support during the development of new learning materials for a Java Programming Language course. This parser was developed using the Eclipse Abstract Syntax Tree framework. This framework generates an Abstract Syntax Tree (AST) that completely represents the program source. AST consists of several nodes, each containing sets of information known as structural properties. For example, Fig. 4 shows the structural properties for the following method declaration:

```
public void start(BundleContext context) throws Exception {
            super.start(context);
}
```



```
⊞ > method binding: Activator.start(BundleContext)
  JAVADOC: null
⊟ MODIFIERS (1)
    ⊟ Modifier [598, 6]
          KEYWORD: 'public'
  CONSTRUCTOR: 'false'
  TYPE_PARAMETERS (0)
⊟ RETURN_TYPE2
    ⊟ PrimitiveType [605, 4]
        ⊞ > type binding: void
          PRIMITIVE_TYPE_CODE: 'void'
⊟ NAME
    ⊟ SimpleName [610, 5]
        ⊞ > (Expression) type binding: void
        ⊞ > method binding: Activator.start(BundleContext)
          Boxing: false; Unboxing: false
          ConstantExpressionValue: null
          IDENTIFIER: 'start'
⊟ PARAMETERS (1)
    ⊞ SingleVariableDeclaration [616, 21]
  EXTRA_DIMENSIONS: '0'
⊟ THROWN_EXCEPTIONS (1)
    ⊟ SimpleName [646, 9]
        ⊞ > (Expression) type binding: java.lang.Exception
          Boxing: false; Unboxing: false
          ConstantExpressionValue: null
          IDENTIFIER: 'Exception'
⊞ BODY
```

🟧 „List of child nodes“ (ChildListPropertyDescriptor)

🟩 „Child node“ (ChildPropertyDescriptor)

🟨 „Simple value“ (SimplePropertyDescriptor)

Fig. 4. Structural properties of a method declaration

Table 1. Sample of JavaParser output

Source	Output

public void start(BundleContext context) throws Exception {
super.start(context);
}
	PublicAccessSpecifier, MethodDefinition, VoidDataType, FormalMethodParameter, ThrowsSpecification, ExceptionClass, SuperReference, SuperclassMethodCall, ExpressionStatement

After building the tree using Eclipse AST API, the parser performs a semantic analysis using the information in each node. This information is used to identify fine-grained indexes for the source program. Table 1 shows the output concepts of JavaParser for the code fragment mentioned above. Note that the goal of the parser is to detect the lowest level ontological concepts behind the code as the upper level concepts can be deduced using ontology link propagation. For example, parser detects "PublicAccessSpecifier" ignoring the upper-level concept of "Modifier".

We compared the accuracy of JavaParser with manual indexing for 103 Java problems and determined that our parser was able to index 93% of the manually indexed concepts. Therefore, an automatic parser can replace the time-consuming process of manual indexing with a high precision and open the way to community-driven problem authoring and targeted expansion in the size of the body of problems.

3 Conclusion

Having fine-grained indexing for programming problems is necessary for better sequencing of learning materials for students; however, the cost of manual fine-grained indexing is prohibitively high. In this paper, we presented a fine-grained indexing approach and tool for the automatic indexing of Java problems. We also explored an application of fine-grained problem indexing during exam preparation, where smaller grain size of knowledge units is critical to finding the sequence of problems which will fill the gaps in student knowledge. Results show that the proposed automatic indexing tool can offer the quality of indexing that is comparable with manual indexing by an expert at a fraction of the cost.

References

1. Brusilovsky, P.: A framework for intelligent knowledge sequencing and task sequencing. In: Proc. of Second International Conference on Intelligent Tutoring Systems, ITS'92. Springer-Verlag (1992) 499-506
2. Brusilovsky, P., Sosnovsky, S., Yudelson, M.: Addictive links: The motivational value of adaptive link annotation. New Review of Hypermedia and Multimedia 15, 1 (2009) 97-118
3. Hosseini, R., Brusilovsky, P., Guerra, J.: Knowledge Maximizer: Concept-based Adaptive Problem Sequencing for Exam Preparation. In: Proc. of the 16th International Conference on Artificial Intelligence in Education. (2013) In Press
4. Hsiao, I.-H., Sosnovsky, S., Brusilovsky, P.: Guiding students to the right questions: adaptive navigation support in an E-Learning system for Java programming. Journal of Computer Assisted Learning 26, 4 (2010) 270-283
5. Kavcic, A.: Fuzzy User Modeling for Adaptation in Educational Hypermedia. IEEE Transactions on Systems, Man, and Cybernetics 34, 4 (2004) 439-449
6. McArthur, D., Stasz, C., Hotta, J., Peter, O., Burdorf, C.: Skill-oriented task sequencing in an intelligent tutor for basic algebra. Instructional Science 17, 4 (1988) 281-307
7. Vesin, B., Ivanović, M., Klašnja-Milićević A., Budimac, Z.: Protus 2.0: Ontology-based semantic recommendation in programming tutoring system. Expert Systems with Applications 39, 15 (2012) 12229-12246
8. Yudelson, M., Providing service-based personalization in an adaptive hypermedia system. PhD Thesis. U. of Pittsburgh, 2010.