Journal of Information & Systems Management

DLINE JOURNALS

ISSN: 2230 – 8776

JISM 2025: 15 (3)
https://doi.org/10.6025/jism/2025/15/3/112-122

A Scaffolding Tool for Generating Java Program Skeletons and Buggy Programs

Ricardo Queirós
CRACS & INESC-Porto LA & DI-ESEIG/IPP
Porto, Portugal
ricardo.queiros@eu.ipp.pt

ABSTRACT

This work introduces CodeSkelGen, a scaffolding tool designed to generate Java program skeletons and buggy programs from annotated solution code provided by teachers. The tool utilizes Java annotations and an annotation processor to create variations of programming exercises, aiding novice students in focusing on problem-solving by reducing the cognitive load associated with writing complete programs from scratch. Skeleton programs provide structural guidance, while buggy programs encourage debugging and testing practices. CodeSkelGen is integrated into an educational framework called Ensemble, which organizes e-learning systems for computer programming education. The generated exercises are packaged using the IMS Common Cartridge standard for reuse across platforms. By automating the generation of exercise materials, including test cases and feedback, CodeSkelGen supports efficient teaching workflows and enhances student engagement through structured practice.

Keywords: Java Program Skeletons, E-learning Systems, Programming Education

Received: 12 November 2024, Revised 6 April 2025, Accepted 22 April 2025

Copyright: with Authors

1. Introduction

Numerous research studies [2, 3] have documented instances where students were assessed on a standard set of programming tasks, revealing that most students performed below expectations. The reasons behind the students' struggles to create the necessary programs remained unclear. One potential reason is that students, especially those who are novices, may not possess a solid understanding of essential programming concepts. Another possibility is that, even if students are familiar with these concepts, they might lack the skills needed to "problem-solve" [6].

One strategy highlighted in these studies involved providing students with skeleton code that they were

required to complete to fulfill the problem's requirements. This method proved effective, and students regarded it as a favorable option. Another approach utilized involved presenting students with faulty programs. In this scenario, students were tasked with identifying logical errors within the program, thereby encouraging skills such as debugging and testing. The reasoning is straightforward: by supplying a skeleton or faulty programs, the "problem-solving" challenge is alleviated, allowing students' working memory to concentrate on forming a new mental model for the problem at hand.

This paper introduces CodeSkelGen, a scaffolding tool designed to generate Java programs from an annotated solution provided by the instructor. The generation mechanism relies on annotations distributed throughout the code. These annotations are formally defined through an annotation type that encompasses all potential actions to be performed in the source code. When the Java source code is compiled, annotations are handled by a compiler plug-in known as the annotation processor. This type of processor generates additional Java source files that represent different versions of the solution program created by the instructor. These versions can be categorized into two kinds: skeleton and buggy.

Skeleton programs facilitate the initial stages of problem-solving by students, enhancing their comprehension of the issues at hand. With the provided structure, students can concentrate on the essence of the problem and abstract their foundational knowledge.

Buggy programs contain logical and/or execution flaws. These programs can encourage students to engage in debugging and testing their code. This practice is often overlooked, leading to poorly functioning programs. The impetus for developing this tool stemmed from the desire to incorporate a code generation feature within an Ensemble instance. Ensemble is a conceptual tool designed to organize networks of e-learning systems and services based on global content and communication standards. Its sole focus is on the teaching and learning process. Within this framework, the emphasis is on coordinating educational services that are commonplace in the daily routines of educators and students, such as the creation, resolution, delivery, and assessment of assignments. The Ensemble instance tailored for the computer programming field relies on the execution of programming exercises to enhance programming skills.

This instance encompasses a collection of components for the creation, storage, visualization, and evaluation of programming exercises, all orchestrated by a central component (teaching assistant) that facilitates communication among all components.

The remainder of this paper is organized as follows: Section 2 presents CodeSkelGen with emphasis on its two components: annotation type and annotation processor. Then, we present an explanation on how to use the annotations formally described in the source code. Then, to evaluate the generation tool, we present its integration on a network for computer programming learning. Finally, we conclude with a summary of the main contributions of this work and a perspective of future research.

2. CodeSkelGen

CodeSkelGen is a code generator tool that generates Java partial programs. The teacher starts by creating the solution program for a specific problem and annotates the code based on the CodeSkelGen annotation type. Upon compilation the Java compiler uses the CodeSkelGen annotation processor to produce several sources

files based on the annotations found in the solution program. The architecture of the generator tool is depicted in Figure 1.

The generated source files can be of two types: skeleton or buggy. In the former some methods are replaced by dummy methods. In the latter, operators and values are exchanged to produce buggy programs (with logic/ execution errors).

2.1 Annotation Type

Annotations, in the Java computer programming language, are a form of syntactic metadata that can be added to Java source code. At runtime, annotations with runtime retention policy are accessible through reflection. At compile time, annotation processors (compiler plug-ins) will handle the different annotations found in code being compiled.

Java defines a set of annotations that are built into the language. The compiler reserves a set of special annotations (including @Deprecated, @Override and @SuppressWarnings) for syntactic purposes. However it is possible to create your own annotations by means of the creation of annotation types. Annotation type declarations are similar to normal interface declarations. An at-sign (@) precedes the interface keyword. Each method declaration defines an element of the annotation type.

In annotation declarations, you can also specify additional parameters, for instance, what types of elements can be annotated (e.g. classes, methods) and how long the marked annotation type will be retained (CLASS – included in class files but not accessible at run-time; SOURCE - discarded by the compiler when the class file is created; and RUNTIME available at run-time through reflection).

In the CodeSkelGen, the interface CSG was created with a set of methods enumerated at Listing 1. The interface is composed by several methods. The most important are:

changeOperator() - replaces the operator (arithmetic, relational or logic) by another;

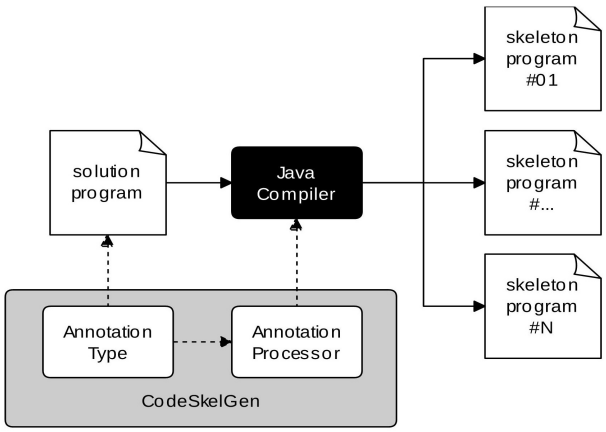

Figure 1. CodeSkelGen Architecture

[1] http://ensemble.dcc.fc.up.pt/

Listing 1 CodeSkelGen Annotation Type

```
package CodeSkelGen ;

      @Retention ( RetentionPolicy . SOURCE )

      public @interface CSG {

      String changeOperator () default "";

      String changeValue () default "";

      String changeVariable () default "";

      String comment () default "";

      boolean removeBody () default false ;
      ...
}
```

• *changeValue()* - replaces a specific value (literal) by another;

• *changeVariable()* - replaces a specific variable by another existent one;

• *changeVariableType()* - change the variable types;

• *removeParameters()* - remove parameters from a method;

• *comment()* - defines generic messages;

• *removeBody()* - removes all the lines of code of an existing method and includes a return

• statement according to the method return type.

• *removeBodySection()* - removes all the lines of code applied to a while instruction or to a conditional instruction;

• *removeRefVariable()* - remove all the instructions that use a specific variable;

In order to process the CSG annotations you need to create an annotation processor.

2.2 Annotation Processor

Starting with Java 6, annotation processors were standardized through JSR 269 and incorporated into the standard libraries. Also the Annotation Processing Tool (apt) was integrated with the Java Compiler Tool

(javac).

The annotation processor will be the responsible to process the annotations found in the source code. Listing 2 shows an excerpt of the foundations of the CodeSkelGen annotation processor.

A processor will "process" one or more annotation types. First, we need to specify what annotation types that our annotation processor will support by using *@SupportedAnnotationTypes* (in this case all) and the version

Then, we need to declare a public class for the processor that extends the *AbstractProcessor* class from the *javax.annotation.processing* package. *AbstractProcessor* is a standard superclass for concrete annotation processors that contains necessary methods for processing annotations.

Inside the main class a *process()* method must be created. Through this method the annotations available for processing are provided. Note that through *AbstractProcessor*, you also access the *ProcessingEnvironment* interface. In the environment the processor will find everything it needs to get started, including references to the program structure on which it is operating, and the means to communicate with the environment by creating new files and passing on warning and error messages. More precisely, with this interface annotation processors can use several useful facilities, such as:

- *Filer* - a filer handler that enables annotation processors to create new files;
- *Messager* - a way for annotation processors to report errors.

The final step to finish the annotation processor is to package and register it so the Java compiler or other tools can find it. The easiest way to register the processor is to leverage the standard Java services mechanism:

1. Package your Annotation Processor in a Jar file;
2. Create in the Jar file a directory META-INF/services;
3. Include in the directory a file named *javax.annotation.processing.Processor*.
4. Write in the file the fully qualified names of the processors contained in the Jar file, one per line.

The Java compiler and other tools will search for this file in all provided classpath elements and make use of the registered processors.

2.3 Program annotation

After the creation of the annotation type and processor one must code the solution program that will use the annotation type previously created. The following excerpt at Listing 3 shows an annotated solution program coded by the teacher for the factorial problem.

Upon compilation the Java Compiler with the help of the registered annotation processors will generate several source files accordingly with the syntax of the annotations found in the source code and the associated semantic in the annotation processor. Listing 4 shows a possible source file.

Note that due to presentation purposes the program generated combines both program types supported by CodeSkelGen (skeleton and buggy).

Listing 2 CodeSkelGen Annotation Processor.

```
Supported Annotation Types ( " CodeSkelGen .CSG" ) @Supported Source Version ( Source
Version . RELEASE_6 )
p u b l i c c l a s s CSGAnnotation Processor extends A bs tr act Pr o ce s s o r { p
u b l i c CSGAnnotation Processor () {
super ( ) ;
}
@Override
public boolean process (Set <? extends TypeElement> annotations, RoundEnvironment
roundEnv) {
// For each element annotated with the CSG annotation
f o r ( Element e : roundEnv . getElements Annotated With (CSG. c l a s s )) {
// Check i f the type o f the annotated element i s not a f i e l d .
// I f yes , return a warning .
i f ( e . getKind () != ElementKind . FIELD) {
processing Env . get Messager ( ) . print Message ( D i a g n o s t i c . Kind .WARNING,
" Not a f i e l d " , e ) ;
continue ;
}
// Define the following variables : name and clazz . String name = capitalize (
e.getSimpleName ( ) . to String ( ) ) ; TypeElement clazz = ( TypeElement ) e. get
Enclosing Element ( ) ;
// Generate a source file with a specified class name. try {
Java File Object f = processing Env . get Filer ( ) .
create Source File ( clazz.get Qualified Name () + " Skeleton") ; processing Env. get
Messager ( ). print Message ( Diagnostic . Kind.NOTE,
" Creating " + f . to Uri ( ) ) ; Writer w = f . open Writer ( ) ;
//Add the content to the newly generated file . try {
Print Writer pw = new Print Writer (w) ; pw . println ( " . . . " ) ;
pw . flush() ;
} finally {
w. close ( ) ;
}
} catch (IOException x) {
processing Env. get Messager( ) . print Message (
Diagnostic . Kind .ERROR, x . to String ( ) ) ;
}
```

```
    }
    returntrue;
```

Listing 3 Program Annotation.

```
public class Program   {
@CSG ( comment = "Calculate   the   factorial   of   the   sum of 2 numbers") public
static void main (String [] args)   {
long num1 = Long . parse Long (args [0]);
long num2 = Long . parse Long (args [1]);
long total = sum ( num1 , num2);
System . out. println ("Factorial   of   " + total + " is " + fact( total));

}
public static long fact(long num) {@CSG (change Value =" >")
if ( num <=1 ) return 1;
else
@CSG ( change Operator )
return num * fatorial( num - 1);

}
@CSG ( comment =" Complete   the   method !",   remove Body = true ) public   static
long   sum ( long   num1 ,   long   num2 )   {
return num1 + num2 ;
} }
```

Listing 4 Skeleton program generated.

```
public   class   Program   {
// Calculate   the   factorial   of   the   sum   of   2   numbers   received   by   stdin
public   static   void   main ( String []   args)   {
long num1   =   Long . parse Long ( args [0]);
long num2   =   Long . parse Long ( args [1]); long   total   =   sum ( num1 , num2 );
System . out. println (" Factorial   of   "   +   total   +   "   is   "   +   fact(
total ));

}
public static long factorial(long num)   {if  (num >1)
return 1; else

return num * fatorial( num + 1);
}
// Complete   the   method !
public static long sum ( long num1, long num2) {return 1;
```

```
}
}
```

3. Integration into an Educational Setting

The motivation for the creation of this tool came from the need to integrate a code generation facility on an Ensemble instance. Ensemble is a conceptual tool to organise networks of e-learning systems and services based on international content and communication standards.

Ensemble is focused exclusively on the teaching-learning process. In this context, the focus is on coordination of educational services that are typical in the daily lives of teachers and students in schools, such as the creation, resolution, delivery and evaluation of assignments.

The Ensemble instance for the computer programming domain relies on the practice of programming exercises to improve programming skills. This instance includes a set of components for the creation, storage, visualisation and evaluation of programming exercises orchestrated by a central component (teaching assistant) that mediates the communication among all components.

Skeleton programs will be generated during the exercises authoring process (Figure 2) in Petcha [4]. Petcha is a teaching assistant component of an Ensemble instance for the computer programming domain. In the authoring process, the teacher fulfils a set of metadata regarding the exercise, codes the correct solution, annotates it, automatically generates skeleton programs and test cases and finally packages all these files in a IMS Common Cartridge (IMS CC) file. An IMS CC object is a package standard that assembles educational resources and publishes them as reusable packages in any system that implements this specification (e.g. Moodle LMS).

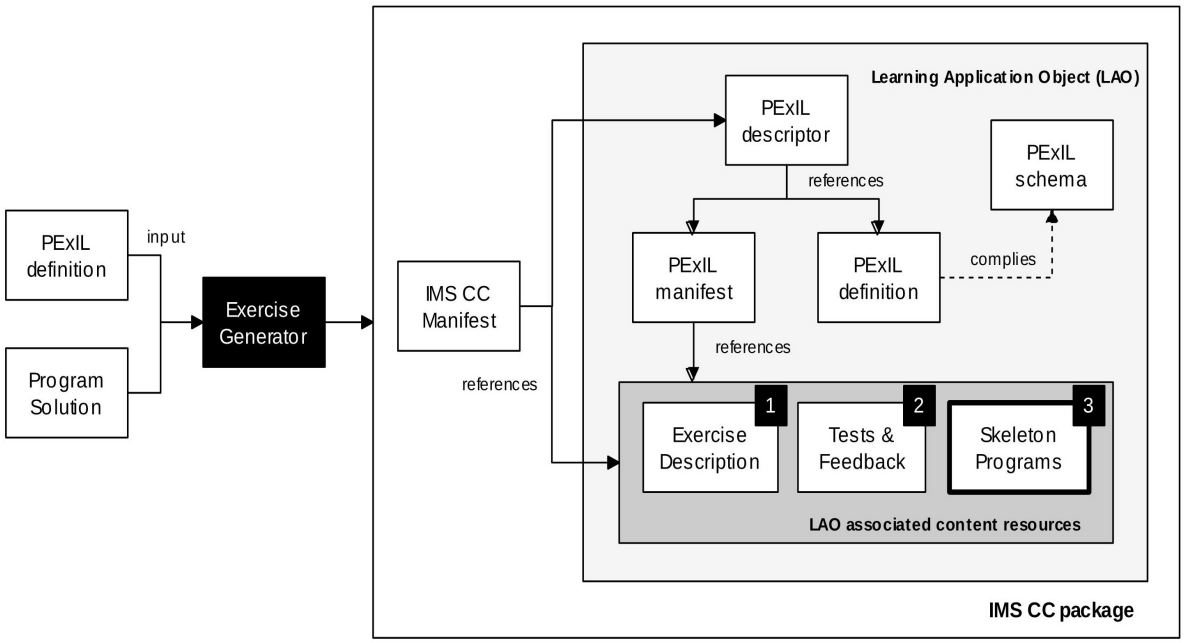

Figure 2. Programming Exercise Package

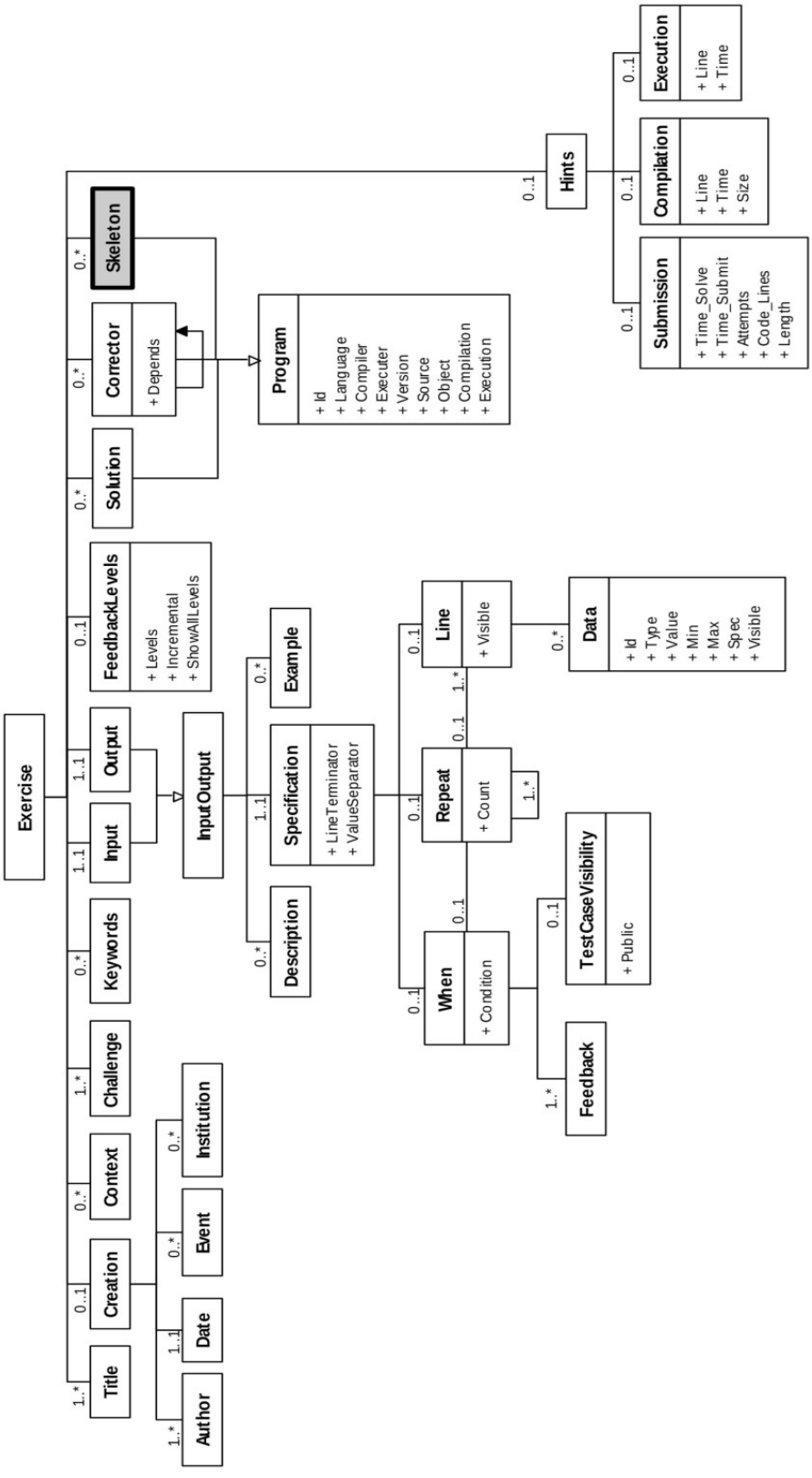

Figure 3. PExIL data model

Since the specification is insufficient to fully describe a programming exercise, an interoperability language was created called PExIL [5]. PExIL describes the life-cycle of a program exercise since its creation until its evaluation. The generation of a learning object (LO) package is straightforward as depicted in Figure 2. The Generator tool uses as input a valid PExIL instance and an annotated program solution file and generates 1) an exercise description in a given format and language, 2) a set of test cases and feedback files and 3) a set of skeleton programs. The PExIL data model depicted in Figure 3 accommodates all these files formalized through the creation of a XML Schema.

The PExIL schema is organized in three groups of elements:

1. Textual - elements with general information about the exercise to be presented to the learner. (e.g. title, date, challenge);

2. Specification - elements with a set of restrictions that can be used for generating specialized resources (e.g. test cases, feedback);

3. Programs - elements with references to programs as external resources (e.g. solution program, correctors, skeleton files) and metadata about those resources (e.g. compilation, execution line, hints).

Then, a validation step is performed to verify that the generated tests cases meet the specification presented on the PExIL instance and the manifest complies with the IMS CC schema. Finally, all these files are wrapped up in a ZIP file and deployed in a Learning Objects Repository (e.g. CrimsonHex [1]).

4. Conclusions and Future Work

This paper presents CodeSkelGen as a code generator tool. Despite not yet implemented most of the design and implementation details were enumerated. The tool is based on annotations. Firstly an annotation type was created to describe the type of operations that can be made on the annotated solution programs provided by the teacher. Secondly, an annotation processor was made to parse these annotations and process them. Finally, an example of how annotate a source file and a possible output was shared to understand the goal of the tool. The tool can produce two types of files: skeleton or buggy (or a combination of both). Based on some studies [2, 3] we think that these types of files will engage novice students on initiating the resolution of exercises and on stimulating them to test more effectively their solutions while using in a regular basis the debugger tools.

The main contribution of this paper is the approach used to generate partial programs. This can be helpful for other people that deal with similar problems. This approach has advantages and disadvantages:

Advantages:

• The processor is external to the source code;

• The annotations processing is at compile time (not runtime);

• The same annotated solution program can be the base for several different versions.

Disadvantages:

• Language dependent (Java);

• Teacher must learn the elements of the annotation type.

As future work, it is expected to implement the CodeSkelGen and enrich the CSG interface with more pertinent constructs. Other research path will be find a language-independent approach to address the main issue of the approach presented in this paper.

References

[1] Leal, J. P., Queirós, R. (2009, May). Crimsonhex: A service-oriented repository of specialised learning objects. In: *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS 2009)* (Vol. 24, pp. 102–113). Springer-Verlag.

[2] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '04)* (pp. 119–150). ACM.

[3] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Ben-David Kolikant, Y., Laxer, C., Thomas, L., Utting, I., Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '01)* (pp. 125–180). ACM.

[4] Queirós, R., Leal, J. P. (2012, July). Petcha – A programming exercises teaching assistant. In: *Proceedings of the 17th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2012)*, Haifa, Israel. ACM.

[5] Queirós, R., Leal, J. P. (2011). Pexil: Programming exercises interoperability language. In *Conferência - XML: Aplicações e Tecnologias Associadas (XATA 2011)*.

[6] Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Kumar, P. K. A., Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO taxonomies. *In*: *Proceedings of the 8th Australasian Conference on Computing Education (ACE '06), Volume 52* (pp. 243–252). *Australian Computer Society*, Inc.