View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Científico do Instituto Politécnico do Porto

CodeSkelGen – A Program Skeleton Generator

Ricardo Queirós

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

Abstract

Existent computer programming training environments help users to learn programming by solving problems from scratch. Nevertheless, initiating the resolution of a program can be frustrating and demotivating if the student does not know where and how to start. Skeleton programming facilitates a top-down design approach, where a partially functional system with complete high-level structures is available, so the student needs only to progressively complete or update the code to meet the requirements of the problem.

This paper presents CodeSkelGen - a program skeleton generator. CodeSkelGen generates skeleton or buggy Java programs from a complete annotated program solution provided by the teacher. The annotations are formally described within an annotation type and processed by an annotation processor. This processor is responsible for a set of actions ranging from the creation of dummy methods to the exchange of operator types included in the source code.

The generator tool will be included in a learning environment that aims to assist teachers in the creation of programming exercises and to help students in their resolution.

1998 ACM Subject Classification D.3 Programming Languages; D.3.4 Processors; Code generation

Keywords and phrases Code Generation, Programming Languages, Annotation

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.145

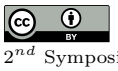
1 Introduction

Several studies [2, 3] reported experiences where the students were tested on a common set of program-writing problems and the majority of students performed more poorly than expected. It was not clear why the students had difficulties to write the required programs. One possible explanation is that students, mainly novice students, lacked knowledge of fundamental programming constructs. Another explanation is that students despite being familiar with the constructs lacked the ability to "problem solve" [6].

One of the approaches mentioned in these studies was the delivery of skeleton code that the students should complete to meet the problem requirements. This approach was validated successfully and students found it a good choice. Other approach used was the definition of buggy programs. In this case the students would have to find logic errors in the program thus stimulating valences as debugging and testing. The rationale is simple: with the delivery of skeleton or buggy programs, the "problem-solving" issue is softened and the students' working memory is free to build a new mental model of the problem to solve.

This paper presents CodeSkelGen as a scaffolding tool to generate Java programs from an annotated solution program provided by the teacher. The generation process is based on annotations scattered throughout the code. These annotations are formally described through an annotation type that includes all the possible actions to make in the source code.

When Java source code is compiled, annotations are processed by a compiler plug-in called annotation processor. This type of processor will produce additional Java source files


© Ricardo Queirós;
licensed under Creative Commons License CC-BY
2nd Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 145–154

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

as versions of the solution program created by the teacher. These versions can be of two types: skeleton and buggy.

Skeleton programs will accelerate the beginning of exercises resolution by the students and facilitate their problem understanding. With the structure included, students can now focus on the core of the problem and abstract their foundations.

Buggy programs include logic and/or execution errors. These type of programs can stimulate students to debug and test their programs. Often this is a forgotten practice which leads to malfunctioning programs.

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[1] 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.

The remainder of this paper is organised 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 on a source code. Then, to evaluate the generation tool, we present its integration on a network for the 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

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

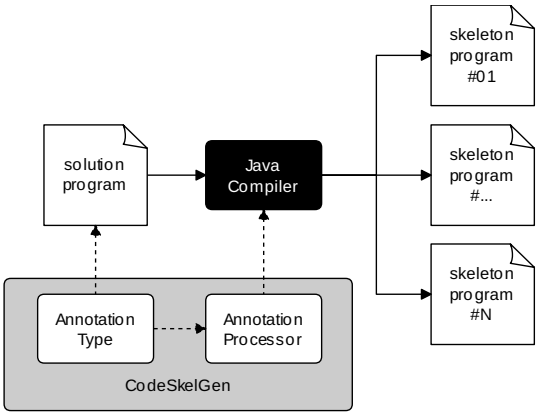


Figure 1 CodeSkelGen Architecture.

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;
    ...
}
```

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;
- *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 of the source files that are supported by using *@SupportedSourceVersion* (in this case the version is JDK 6).

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.

```
SupportedAnnotationTypes( "CodeSkelGen.CSG" )
@SupportedSourceVersion( SourceVersion.RELEASE_6 )
public class CSGAnnotationProcessor extends AbstractProcessor {
 public CSGAnnotationProcessor() {
  super();
 }
 @Override
 public boolean process(Set<? extends TypeElement> annotations,
    RoundEnvironment roundEnv) {
  //For each element annotated with the CSG annotation
  for (Element e : roundEnv.getElementsAnnotatedWith(CSG.class)) {
   //Check if the type of the annotated element is not a field.
   //If yes, return a warning.
   if (e.getKind() != ElementKind.FIELD) {
    processingEnv.getMessager().printMessage(Diagnostic.Kind.WARNING,
        "Not a field", e);
    continue;
   }
   //Define the following variables: name and clazz.
   String name = capitalize(e.getSimpleName().toString());
   TypeElement clazz = (TypeElement) e.getEnclosingElement();

   //Generate a source file with a specified class name.
    try {
        JavaFileObject f = processingEnv.getFiler().
            createSourceFile(clazz.getQualifiedName() + "Skeleton");
        processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE,
            "Creating " + f.toUri());
        Writer w = f.openWriter();
        //Add the content to the newly generated file.
        try {
            PrintWriter pw = new PrintWriter(w);
            pw.println("...");
            pw.flush();
        } finally {
            w.close();
        }
    } catch (IOException x) {
        processingEnv.getMessager().printMessage(
                        Diagnostic.Kind.ERROR, x.toString());
    }
   }
   return true;
  }
}
```

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.parseLong(args[0]);
  long num2 = Long.parseLong(args[1]);
  long total = sum(num1,num2);
  System.out.println("Factorial of " + total + " is " + fact(total));
 }
 public static long fact(long num) {
  @CSG(changeValue=">")
  if (num <=1 )
    return 1;
  else
    @CSG(changeOperator)
  return num * fatorial(num - 1);
 }
 @CSG(comment="Complete the method!", removeBody=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.parseLong(args[0]);
  long num2 = Long.parseLong(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).

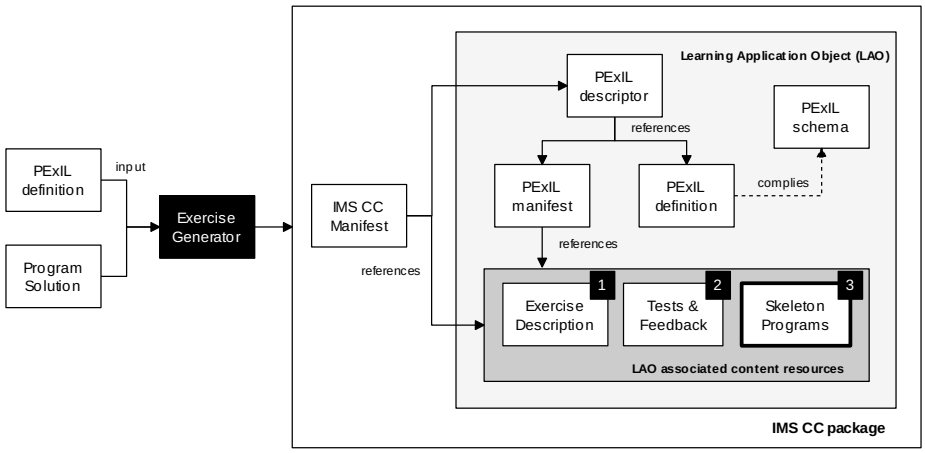


Figure 2 Programming Exercise Package.

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.

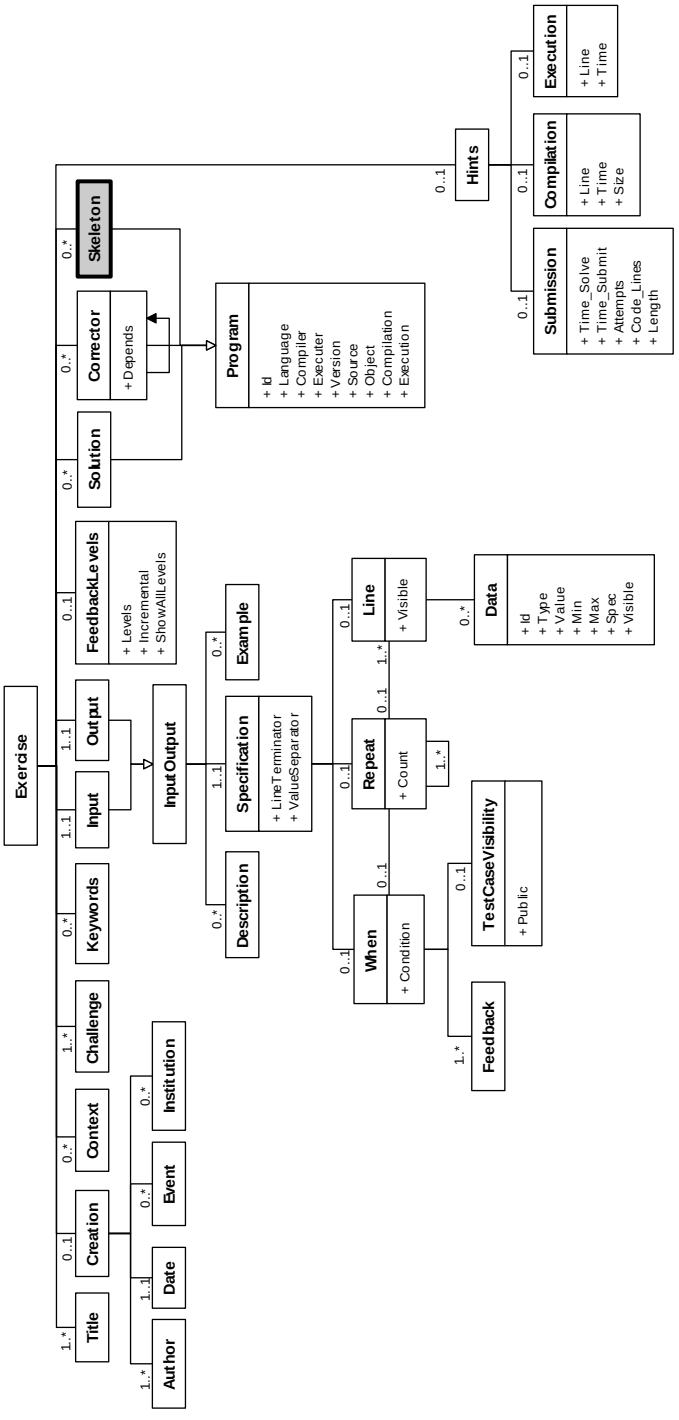


Figure 3 PExIL data model.

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 José Paulo Leal and Ricardo Queirós. Crimsonhex: a service oriented repository of specialised learning objects. In *ICEIS 09 - 11th International Conference on Enterprise Information Systems, Milan, Italy*, volume 24 of *Lecture Notes in Business Information Processing*, pages 102–113. Springer-Verlag, LNBIP, Springer-Verlag, LNBIP, May 2009.

2 Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 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, pages 119–150, New York, NY, USA, 2004. ACM.

3 Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 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, pages 125–180, New York, NY, USA, 2001. ACM.

4 Ricardo Queirós and José Paulo Leal. Petcha - a programming exercises teaching assistant. In *ACM SIGCSE 17th Anual Conference on Innovation and Technology in Computer Science Education*, Haifa, Israel, July 2012 2012. ACM.

5 Ricardo Queirós and José Paulo Leal. Pexil: Programming exercises interoperability language. Conferência - XML: Aplicações e Tecnologias Associadas (XATA), 2011.

6 Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 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 - Volume 52*, ACE '06, pages 243–252, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.