Architectural Model for Generating User Interfaces Based on Class Metadata

Luiz Azevedo, Clovis Fernandes, Eduardo Guerra

Aeronautical Institute of Technology, Praça Marechal Eduardo Gomes, 50
Vila das Acacias, CEP 12.228-900 – São José dos Campos – SP, Brazil
{ luizfva, guerraem}@gmail.com
clovistf@uol.com.br

Abstract. Source code duplication is the origin of several problems in a software development project. Even been aware of this situation, application developers tend to ignore them, once it takes a lot of time and effort for duplicated pieces of code to be found and eliminated. To address this issue, the present work presents a new model for source code generation for user interface development. The generation process happens at runtime, each time a page is requested, uses the resulting content of the request processing and a set of templates and is based on class metadata. As result, application developers have new tool to avoid inconsistencies that can be originated by code duplication.

Keywords: user interface, source code generator, metadata, framework, inconsistency, software architecture.

1 Introduction

Source code duplication is an evil we all must fight. There are times when we feel we don't have a choice, the application environment seems to require duplication, or we simply don't realize we are duplicating information [1] and that can negatively affect the software development in many ways.

Code duplication results in increased code size and complexity, making program maintenance more difficult. Even if programmers could find and edit each copy of a piece of code, it is impossible to ensure that all the changes were made consistently – that the common regions are identical and the differences are retained – without manually comparing each clone's body, word-by-word, and hoping that no important details were missed [2].

Whatever the reason, there are ways to avoid replicating source code. Source code generation is a well-known tool to prevent a piece of information to be spread out to several places of an application [1,3].

Structures in multiple languages can be built from a common metadata representation using a simple code generator each time the software is built [1]. Particularly, when dealing with User Interfaces (UI), another approach would be to use a single source of metadata in order to generate the source code at runtime.

The purpose of this paper is to present the model for source code generation for user interface development. It is divided in the following manner: Section 2 introduces inconsistencies in user interfaces, Section 3 presents the categories of source code generators, Section 4 introduces the proposed model, Section 5 presents a framework created as a proof of concept, Section 6 reports the validation of this work and Section 7 presents the conclusions.

2 Inconsistency in User Interfaces

At a survey conducted in early 1990s, Myers and Rosson [4] found that up to 48% of software application source code is devoted to the user interface portion. Though there are no recent studies to verify whether this relation stays the same, a considerable part of the software development time is still dedicated to the user interface development. On that occasion, interviewees reported that achieving consistency was one of the most common faced problems when developing a user interface, especially when there are multiple developers involved [4].

The lack of consistency might be a symptom of potential source code duplication but, when it comes to user interfaces, it is rarely treated as such. Focusing on a web application, three different kinds of inconsistencies, involving user interfaces, were identified.

Inconsistencies between HTML code and application code

When listing the good habits of a pragmatic programmer, Hunt and Thomas [1] highlight the Don't Repeat Yourself (DRY) principle as: *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system*.

While developing a user interface, a common, yet bad, practice is to explicitly express in it every piece of information about the content that will be presented to the final user. By doing so, the DRY principle is been violated, i.e. not only the `Student` class knows that a student has a `name`, but every page that presents information about a student also does. Thereby, when it comes the time a student's name is no longer stored in a single field, instead it must be stored by the student's first-name and last-name, all those pages that displayed the student name will have to be changed. If, by any chance, at least one page is not updated, an inconsistent state arises. This kind of inconsistency is usually found in applications that separate user interface code from application logic, like MVC (Model-View-Controller) applications.

This is the simplest kind of inconsistency. The support provided by integrated development environments (IDE) usually helps to avoid, or at least helps to identify this kind of inconsistency. When they don't, the first time someone navigates into an inconsistent page, an error message will be printed out.

Inconsistencies in the way a content type is presented

Sometimes, it is desirable that a type of information always be presented the same way (consistently) to the final user. For example, a `java.util.Calendar` object usually holds more information than the user would want to see, thus it is customary to format its date in a way the user could read its information and the date formatting pattern should be the same throughout the whole application.

By violating that pattern, the application usability may be compromised. Besides, it is evidence that the formatting code is been duplicated among the pages of the application.

Behavioral inconsistencies among different user interfaces

When a graphical element has some events associated with it, like an animation or client-side content validation, it should trigger these events in the same manner, independently from what content it holds or by what page it is been used.

Suppose that all mandatory fields for text entry have a minimum length to be considered valid. For example, in a form for adding students an input value like "a" for the student name field should not be considered valid. If this validation is not performed in a mandatory text field, inconsistent data can be informed and persisted to the application.

3 Source code generation

Source code generators are great tools to reduce the odds that fragments of source code be copied to several places. Source code generation is about writing programs that write programs [3]. According to Herrington [3] the benefits of the use of a source code generator are:

- Quality: large volumes of manually written source code tend to have inconsistent quality, because new and better approaches are found as the application is been developed, however, by several reasons, they cannot be applied to every portion of the application that has already been developed.
- Consistency: The source code that is built by a generator is consistent in the design decisions and conventions. That is, one can't be certain that design decisions and source code conventions will be respected when the source code is written manually. However, when a generator acts as an interface between two distinct contexts of an application, like between a database and the data access layer or between the model layer and the presentation layer, consistency is always preserved, for one context is built in according to the other.
- Abstraction: There is the possibility of creating new templates to translate the logic into other languages, onto other platforms, or programming paradigms.

- Productivity: The work time of a software engineer is highly valuable and should not be wasted with repetitive and predictable activities. The generator can treat those repetitive activities, and then the engineer can apply his efforts to a more noble activity.

Source code generators are classified into two categories as for its influence over the built code: active and passive [3]. Passive generators build source code that can be freely changed, that is, developers have full access to the content created and when necessary, it can be directly modified. Hence, when the generation is performed again, all the modifications made manually will be lost [5]. This type of code generators is usually used by IDEs for creating new files with a starting content.

Active source code generators, on the other hand, are responsible by the code generated. It is important to highlight that active code generators use templates as basis of the creation process [6]. As the need to modify the code arises, they are made to those templates, instead of to the resulting code, and the changes are reflected on the created content. Active source code generators can be used along with the build process or at runtime as the proposed model.

There is also a classification as for the output produced. Herrington [3] describes them as follows:

- Code Munger Generator: this kind of generator takes an input file, usually source code, and searches it for patterns. When it finds those patterns, it processes their contents and generates a set of one or more output files. A well-known example of a Code Munger Generator is the JavaDoc documentation generated from a set of java source files.
- Inline-code Expansion Generator: simplifies the source code by adding a specialized syntax, in witch you specify the requirements for the code to be generated. This syntax is parsed by the generator, which then implements code based on requirements. The source code is "expanded" based on some kind of markup, which will be replaced by the generated code. The output file will have all the original source code, except for the special markup that is going to be replaced by the built content.
- Mixed-code Generator: The generator reads the input file and modifies it, overwriting the original file with the changes made. As the Inline-code Expansion Generators, the Mixed-code Generators also use special markups that indicate where the generated code will be placed. The main difference is that the Mixed-code Generators keep the original markup that will denote where the generate code was placed.
- Partial-class Generator: The generator uses an abstract definition of the source code to be created as input. Instead of filtering, or replacing code fragments, this generator takes a description of the code to be created and builds a full set of implementation code.
- Tier Generator: The generator builds all the code for one layer or section of an application. The most common example is the constructor of a database access layer of a web/client-server application.

4 The Proposed Model

This research proposes a new model of source code generators for user interface development, called MAGIU – Architectural Model for Generating User Interfaces (in Portuguese, *Modelo Arquitetural de Geração de Interfaces com o Usuário*), it can be implemented in any object oriented language and guarantees the consistency of an application user interfaces. The proposed model suits the characteristics of an active, inline-code expansion generator [3,6,5], its parameterization and generations are made at runtime.

It has been chosen to use an active source code paradigm by the possibility to customize the resulting content by acting directly over the generation process, once the templates used by the generator are passive to changes. On the other hand, the inline-code expansion paradigm was chosen so that the special markup that indicates where the generated content will be placed is deleted and will not impact the resulting content.

For the sake of simplicity, from now on, all the given examples will be for an HTML code generator and the server-side language used will be Java.

The generated content corresponds to the HTML code necessary to present or to edit server-side objects' state, and it will be placed at an HTML page each time the user requests it. Those server side objects are usually the resulting content of a calculation or a query to the database, whose metadata are read and then used to build HTML code. The resulting HTML content corresponds to an editor (an HTML form) or a display for server-side object.

The proposed model has three main sources of information in the process of building the user interface:

- Metadata: the generator reads metadata information from an instance object. This object contains the primary source of metadata consumed by the generator besides the information that will be show to the user or edited by him.
- Templates: define the organization of the HTML code, which will compose the generated user interface. Each template is directly related to a specific type of data (as templates for texts, for boolean data, for numbers or user defined data). Application developers can create new templates as well as edit the existing ones in order to change the resulting content.
- Metadata Repository: stores the metadata read from various sources, except for that from the instance object given to the generator. The metadata repository avoids unnecessary metadata readings [7].

Figure 1 presents the main elements of the proposed model.

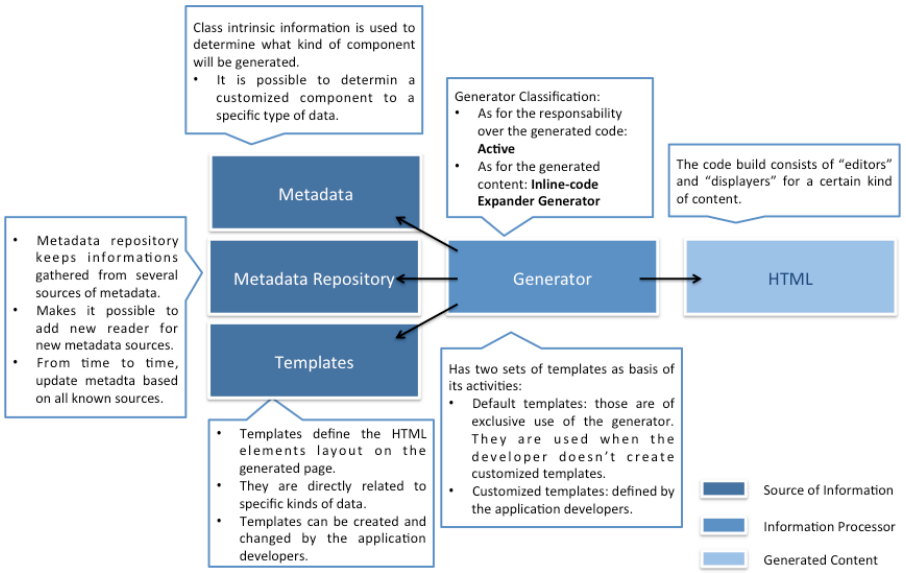

Figure 1 - Proposed Model for generation of source code for user interfaces.

The metadata reading and metadata processing are two of the most important stages of the generation. Characteristics of the generated content vary from application to application and what determine these variations are the different forms that the data is structured. By processing the metadata, the generator is capable of obtaining any information regarding the structure of all necessary data to conclude the generation.

MAGIU model supports the generation of HTML code for two different contexts: content edition and content display. For this, the generator must have access to an object whose content will be displayed or edited. Usually, this object is an instance of an application entity. In practice, any type of object that stores information can be used as a source of knowledge for generating a page.

The model enables the addition of new metadata sources, what makes it possible for the generator to read metadata from any source, even those not foreseen by the moment of its creation.

When an object is provided, the generator searches for a user-defined template related to the type of that object. The lack of a customized template implies in the use of the default template for that corresponding type. If no template is found so far, the generator will search for the template of the immediate supertype of that object. The search continues until it reaches the `Object` class. The implementation should provide default templates for primitive types (such as integers, booleans, floating numbers etc.), text types (string), collection of objects (like the Java `java.util.Collection` interface and arrays) and the `Object` class.

It is also possible that one type of data can represent more than one type of information, i.e. a string object can represent a plain text, a password, an email address, a URL, an image path, and so on. Therefore, the application developer should be able to

add a special markup to add semantics to those types and create a new template for each of these semantic types.

The generator iterates over the fields of the given object repeating the generation processing for each of them. This divide and conquer approach is repeated until a type of interest is found. On the context of this work, we define a type of interest, as been an atomic type of data. Once divided, a type of interest has no meaningful information to the final user. A common example is a `java.util.Calendar` object, whose fields might not bring useful information besides the actual date that the final user would be interested in. In Java, some examples of types of interest are `java.lang.String`, `java.util.Calendar`, `java.lang.Number`, `java.lang.Character` and any of their subclasses plus the java primitive types.

Generating the source code for a type of interest is the base case of this recursive algorithm. Every type of interest has a correspondent template for display's generations and other for editor's generations. Once the HTML code is built, the generation process goes up a level.

Figure 2 and Figure 3 illustrate the recursive generation algorithm defined by the proposed model.

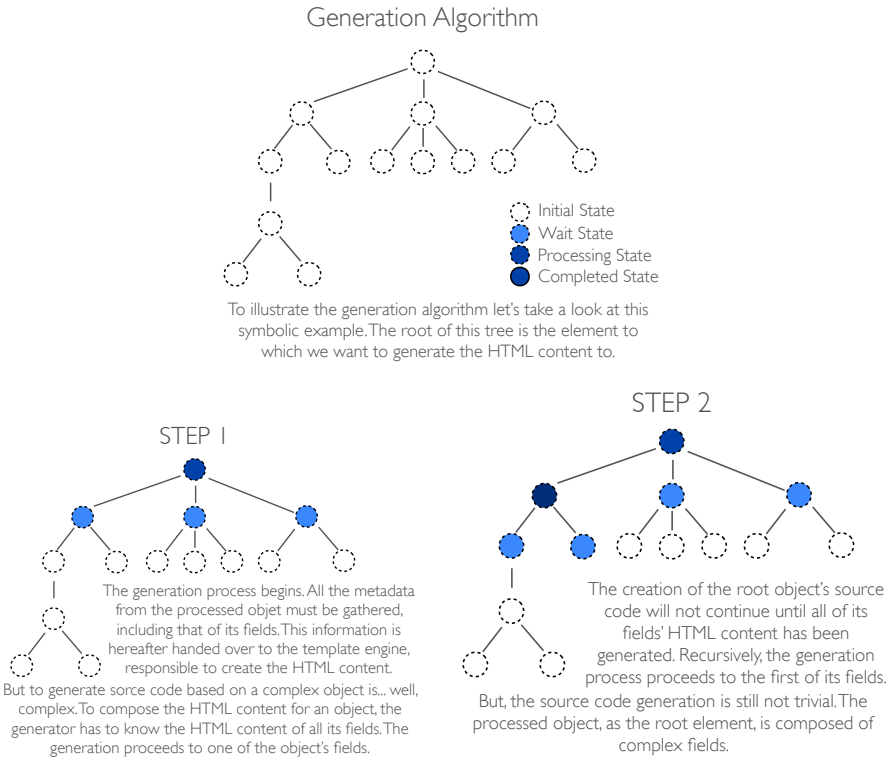

Figure 2 - Generation Algorithm (1/2)

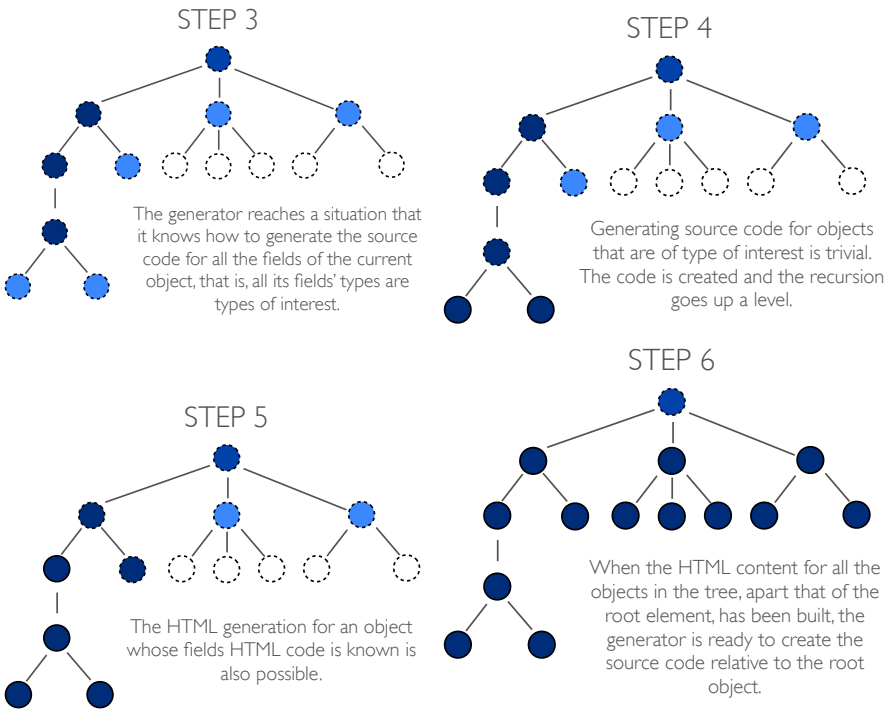

Figure 3 - Generation Algorithm (2/2)

The MAGIU model helps to avoid the three kinds of inconsistencies introduced previously:

1. Inconsistencies between HTML code and application code: the HTML code is always generated from the application code;
2. Inconsistencies in the way the content is presented: a given type of application code is always presented the same way, that is, their generation process and their template are the same;
3. Behavioral inconsistencies among different user interfaces: the event calls originated by a kind of HTML element is consistent throughout the application, once that element is always generated from the same template.

However, the developers have full access to the templates, so they can introduce inconsistencies into them. It has been chosen to make customization possible, so that developers could achieve the desired user interface, but as a drawback, it is not possible to check whether the templates are consistent with each other.

5 Development Framework

A Java development framework was built, as a proof of concept, to show the real capacity of the proposed model of source code generation. According to Foote and

Johnson [8], a framework is a set of classes that embodies an abstract design for solutions to a family of related problems. It consists of a structure that can be reused as a whole on the development of a new system [9].

All interaction with the framework is done through its façade. The client applications make calls to two methods, one to generate an editor for given a content and one to generate a display for a given content.

MAGIU public interface

```
public class MAGIU {
        public String editorFor(Object model) {…}
        public String displayFor(Object model) {…}
}
```

The metadata read is encapsulated in a Metadata Container object [7] and given to a template engine. The templates have some small regions that will be filled with information provided in that metadata container.

The following example illustrates a template used to edit the content of an object of type `Object`. The Object template is taken as the default template for complex objects. The Properties collections represents the collection of metadata gathered from all the instance variables that object has, accessible through the Metadata Container objects. This collection will be iterated, generating HTML content for every field.

Example of template for edit an Object's content.

```
<div id="${DisplayName}">
  <#list Properties as prop>
    <p>
      <label class="inFieldLabel"
             for="${prop.nameAttribute}">
        ${prop.displayName}
      </label>
      ${prop.html}
    </p>
  </#list>
</div>
```

Accessing the `prop` variable, the template engine can print every information hold by the related Metadata Container, they are:

- `nameAttribute`: metadata usually used by templates of types of interest. It is used in the "name" attribute of some HTML tags. This way, the HTML form submitted back to the server can be easily bound to an object.
- `displayName`: a display name that the current object/field has in a form or in a display page.
- `templateName`: the name of a template, used to specify the usage of a specific template.
- `value`: a reference to the value of the current object/field. When used in a template, its `toString` method is called.
- `type`: a reference to the class of the current object/field.

- `html`: the generated HTML content for the current field.

A common example of usage of a user interface generator is to build CRUD pages. Figure 4 illustrates a form for adding new users generated by the framework.

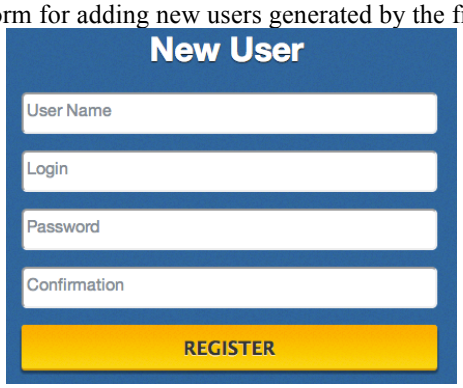

Figure 4 - Example of New User Form, created by a MAGIU generator

The following example shows an example of how a MAGIU generator can be used in a JSP page. The `user` object is attached in the request by the servlet and handled to the generator. The `editFor()` call will then return a String object containing the resulting HTML content.

Example of the usage of the generator in a JSP page.

```
<%@page import="atarefado.view.Html"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags"%>
<%@ taglib prefix="html" uri="/WEB-INF/functions.tld"%>
<jsp:useBean id="user"
  type="atarefado.model.presentationmodel.UserPM" scope="request" />
<my:base title="NewUser">
  <form action="NewUser" method="post">
    <div class="centralized">
      <h1>New User</h1>
      ${html:editorFor(user)}
      <input type="submit" value="Register"
             class="goldButton" />
    </div>
  </form>
</my:base>
```

The UserPM class declaration defines the metadata processed while the form presented in Figure 4 was generated.

UserPM class declaration.

```
public class UserPM {
  @DisplayName("User Name")
  private String name;
  private String login;
  @TemplateName("Password")
  private String password;
  @TemplateName("Password")
  @DisplayName("Confirmation")
  private String passwordConfirmation;
}
```

The templates used in this generation are the Object Template, the String Template and the Password Template.

String Template

```
<input id="${model.nameAttribute}" type="text" class="text"
       name="${model.nameAttribute}">
  ${model.html}
</input>
```

Password Template

```
<input id="${model.nameAttribute}" type="password" class="text"
       name="${model.nameAttribute}">
  ${model.html}
</input>
```

6 Experiment

Bosh [9] highlights that tests must be made to verify whether the framework provides the wanted functionality and to assess its usability. Hence, an experimental study was elaborated to verify if the MAGIU model, implemented in the presented framework, really removes the potential inconsistencies in the user interfaces of an application, as proposed.

The study was conducted with undergraduate students of the Aeronautical Institute of Technology (ITA – Instituto Tecnológico de Aeronáutica). The main goal of the experiment was to verify which kind of inconsistency could be inserted in applications with and without the use of the proposed model. In this context, the students were responsible to perform a modification over a sample application in order to try to intentionally cause inconsistencies.

The artifacts used on the study were written in Java and, even though some participants had a limited experience in developing software, all of them had solid knowledge in Java and were capable to read and comprehend the application logic. Most of the participants had only had developed software at class and their knowledge was strictly academic. Besides, a few students had already had some previous experience involving source code generators. To those students that participated in the studies, the experiment activities came to replace one of the discipline's practical evalua-

tions. Figure 5 illustrates the participants' previous experience with software development.

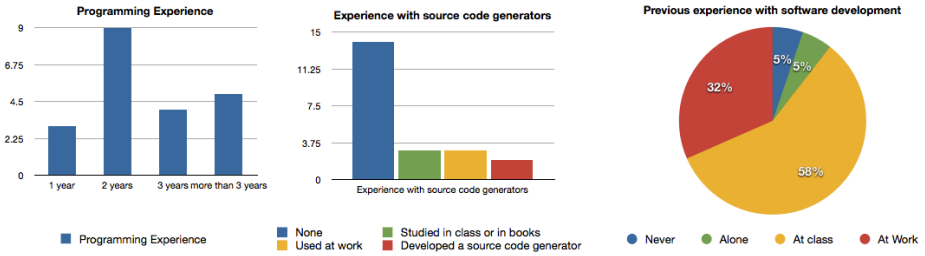

Figure 5 - Participants previous experience with software development

The experiment was divided in three phases: training, implementation and analysis. In the first one, the students went through a training process at which they were presented to the types of source code inconsistencies, its causes, and the usage of the MAGIU generator framework. Additionally, they were also exposed to two versions of an sample application called *Atarefado*, where one used the generator and the other didn't.

In the second phase, the students were divided into ten groups of two or three people and each group chose a piece functionality to be implemented. Their activities consisted of trying to cause one or more inconsistency cases, stressing the MAGIU generator in every imaginable way. Initially, just the version of the *Atarefado* sample application that didn't use the generator should be used (V1). If an inconsistency was found, the group should try to cause that same inconsistency to the version of *Atarefado*, supported by the generator (V2), and report the results. At the end of the experiment, the students answered a survey questionnaire to register their experience.

In the third phase, the reported information, the questionnaire answers and the group's resulting implementation were qualitatively analyzed in the attempt to identify evidences that proved that the generator eliminated the provoked inconsistencies. The main advantage of performing a qualitative analysis is that it demands that the researcher unveil the problem complexity instead of abstracting it [10]. Hence, it is possible to analyze the elements responsible for causing the inconsistencies found by the experiment participants.

Six groups reported inconsistencies in version V1 along the implementation of the chosen piece of functionality as ilustrated in Figure 6. Three of them reported that inconsistencies were also found in version V2 of the *Atarefado* sample application. After the analysis was completed, it was found that all reported inconsistencies were directly inserted on the templates. Those inconsistencies emerged when two or more templates were inconsistent with each other, the CSS classes or JavaScript functions used by one template were not necessarily used by the other.

Thus, inconsistencies found on the templates were replicated to all the pages that used them. However, to solve it, the application developer needs only to correct the inconsistent template, therefore modifying only one file. It is noteworthy that the reported of inconsistency can only happen within the templates created by the appli-

cation developers. By using a consistent set of templates, the generator will always create inconsistency free content.

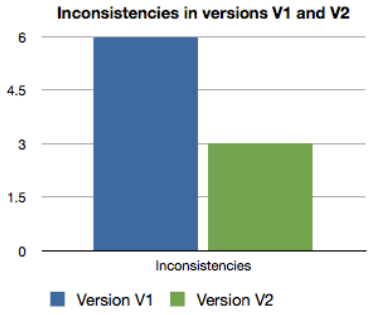

Figure 6 - Reported inconsistencies

7 Conclusions

At the beginning of this research, it was defined that consistency between two objects is the fulfillment of all the rules/relations that bind them. By breaking one of those rules, an inconsistent state is characterized. When the value of a metadata is changed, some rule is also changing, and the generator will always follow that. With no support of a generator like MAGIU generators, it is the application developers' responsibility to know and follow those rules. On the other hand, when a MAGIU generator is adopted, the rules can be abstracted once they will be followed by the generator.

After concluding the analysis of the experiment data, we feel that our hypothesis gained strength. The usage of a MAGIU generator can help the application developers to avoid source code duplication and, consequently, avoid inconsistencies among the pages of an application.

References

1. Andrew Hunt and David Thomas, *The Pragmatic Programmer: From Journeyman to Master*.: Addison Wesley, 1999.
2. Michael Toomim, Andrew Begel, and Susan L. Graham, "Managing Duplicated Code with Linked Editing," in *Visual Languages and Human Centric Computing, 2004 IEEE Symposium*, Rome, 2004, pp. 173-180.
3. Jack Herrington, *Code Generation in Action*. Greenwich, CT, USA: Manning Publications Co, 2003.
4. B. Myers and M. Rosson, "Survey on User Interface Programming. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems," *ACM Press*, pp. 195-202, 1992.

5. Michael Montero. (2009, Maio) The Input Ouput Toolkit - IOTK. [Online]. http://iotkfw.com/2009/05/21/active-passive-code-generation/
6. Autumn Wilkins and Cody Smith. (2011, Julho) CodeSmith Generator 6.x. [Online]. http://docs.codesmithtools.com/display/Generator/Active+vs.+Passive+Generation
7. Eduardo Guerra, "A Conceptual Model for Metadata-based Frameworks," Aeronautics Institute of Technology, São José dos Campos, 2010.
8. Brian Foote and Ralph Johnson, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. I e II, pp. 22-35, Julho 1988.
9. Jan Bosh, Peter Molin, Michael Mattson, and PerOlof Bengtsson, "Object-Oriented Framework - Problems and Experiences," Department of Computer Science and Business Administration, University of Karlskrona, Ronneby, Sweeden, 1997.
10. Forrest Shull, Janice Singer, and Dag Sjøberg, *Guide to Advanced Empirical Software Engineering*. New York: Springer-Verlag, 2007.
11. Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *Proceedings of the 21st international conference on Software engineering*, New York, NY, 1999.
12. Walter L Hürsch and Cristina Videira Lopes, "Separation of Concerns," February 1995.