Article

TagML—An Implementation Specific Model to Generate Tag-Based Documents

Ricardo Tesoriero , Gabriel Sebastian and Jose A. Gallud

Escuela Superior de Ingeniería Informática de Albacete, Universidad de Castilla-La Mancha, 02071 Albacete, Spain; gabriel.sebastian@uclm.es (G.S.); jose.gallud@uclm.es (J.A.G.)

* Correspondence: ricardo.tesoriero@uclm.es; Tel.: +34-967-599-200

Received: 2 June 2020; Accepted: 3 July 2020; Published: 5 July 2020

Abstract: This article describes TagML, a method to generate collections of XML documents using model-to-model (M2M) transformations. To accomplish this goal, we define the TagML meta-model and the TagML-to-XML model-to-text transformation. While TagML models represent the essential characteristics of collections of XML documents, the TagML-to-XML transformation generates the textual representation of collections of XML documents from TagML models. This approach enables developers to define model-to-model transformations to generate TagML models. These models are turned into text applying the TagML-to-XML transformation. Consequently, developers are able to use declarative languages to define model-to-text transformations that generate XML documents, instead of traditional archetype-based languages to define model-to-text transformations that generate collections of XML documents. The TagML model editor as well as the TagML-to-XML transformation were developed as Eclipse plugins using the Eclipse Modeling Framework. The plugin has been developed following the Object Modeling Group standards to ensure the compatibility with legacy tools. Using TagML, unlike other previous proposals, implies the use of model-to-model transformations to generate XML documents, instead of model-to-text transformations, which results on an improvement of the transformation readability and reliability, as well as a reduction of the transformation maintenance costs. The proposed approach helps developers to define transformations less prone to errors than using the traditional approach. The novelty of this approach is based on the way XML documents are generated using model-to-model transformations instead of traditional model-to-text transformations. Moreover, the simplicity of the proposed approach enables the generation of XML documents without the need for any transformation configuration, which does not penalize the model reuse. To illustrate the features of the proposal, we present the generation of XHTML documents using UML class diagrams as input models. The evaluation section demonstrates that the proposed method is less prone to errors than the traditional one.

Keywords: tag languages; meta-model; M2T transformation; M2M transformation

1. Introduction and Background

The number of user interface description languages that use markup languages has increased during the last years. The main reason behind their growth is their use in the Web.

The Extensible Markup Language (XML) [1] is one of the most popular markup languages. For instance, the Extensible HyperText Markup Language (XHTML) [2] enables developers to define static HyperText Markup Language (HTML) [3] pages as valid XML documents.

The use of XML is not limited to the definition of static Web pages, most technologies supporting the generation of dynamic Web pages use XML-based languages; for instance, Java Server Pages (JSP) [4], Java Server Faces (JSF) [5], Standard Tag Library for JavaServer Pages (JSTL) [6], ASP.NET

Web Pages [7], Extensible Application Markup Language (XAML) [8], Polymer components [9], AngularJS directives [10], Extensible 3D (X3D) [11], and so forth.

The use of XML to define user interfaces is not limited to the Web; it also used to define user interfaces in mobile platforms such as the Android [12] platform. It is neither limited to the description of UIs, the XML language is also employed to represent the information exchanged in Web Services. For instance, the Simple Object Access Protocol (SOAP) [13] defines the Web service communication using XML documents.

Another example that exposes the versatility of XML to represent information is the definition of the XML Model Interchange (XMI) [1] format which is the standard format defined by the Object Management Group (OMG) [14] that enables the use and manipulation of model information in Model-driven Architectures (MDAs) [15].

An interesting capability of XML is definition of XML meta-data in XML documents. For instance, the XML Schema Definition (XSD) [16] language is an XML-based language to specify the structure of valid XML documents. Besides, the XML Stylesheet Language Transformation (XSLT) [17] language enables developers to define XML document transformations in XML.

Model-driven Architectures (MDAs) enables developers to generate Platform Specific Models (PSMs) using Platform Independent Models (PIMs) as input parameters of model-to-model transformations. These PSMs are used as input parameters of model-to-text transformations to generate the source code of the application.

Most popular model-to-text transformation languages tend to adopt an archetype-based paradigm. Some examples of this type of language are ACCELEO [18], MOFScript [19], Java Emitter Templates (JET) [20], and so forth. These languages use text templates interwoven with OCL expressions. While the text templates generate static text; the OCL expressions generate dynamic text. They enable developers to define transformations any language or purpose.

This article describes a proposal that captures XML characteristics to enable developers to use model-to-model transformation languages to generate collections of XML documents. Unlike most popular model-to-text transformation languages, model-to-model transformation languages, such as Query View Transformation (QVT) [21] or Atlas Transformation Language (ATL) [22], tend to adopt a declarative paradigm where rules are defined using OCL expressions.

The level of abstraction is leveraged introducing the TagML meta-model to capture the essential properties of XML to model XML documents. In addition, the level of reuse is improved by defining the TagML-to-XML model-to-text transformation that generates collections of XML documents using the TagML models.

The novelty of this approach lays on the way XML documents are generated using model-to-model transformations instead of traditional model-to-text transformations. Consequently, junior developers do not have to learn another programming paradigm to write model-to-text transformations to generate this kind of documents reducing costs (it is assumed that they already have skills in model-to-model transformation languages which usually employ declarative programming paradigms) [23]. Moreover, the simplicity of the proposed approach enables the generation of collections of XML documents without the need for any transformation configuration. This fact does not penalize model reuse due to the proposed generation process.

Our proposal enables developers to generate collections of XML documents using model-to-model transformations. To carry out this task, we encourage the interoperability of the architecture at design time following the MDA principles of leveraging the level of abstraction and reuse [24].

In Reference [25], the authors argue for the importance of integrating XML-based models in the MDE process. In that article, the authors propose a lightweight approach for providing first-class support for managing XML documents within Epsilon model management language family [18] using the EOL [26] OCL-based imperative language to manipulate models. The philosophy behind our approach is the definition of an Implementation Specific Model (ISM) to deal with XML document generation using a declarative programming paradigm to make this process as simple as possible for

junior developers (e.g., our approach does not require Epsilon to work). Though the goal and purpose of both approaches may be similar, the perspective to board the problem is completely different (i.e., imperative versus declarative paradigm to define model transformations).

Main difference between the study presented in Reference [27] and our approach is the level of abstraction they use to address similar problems. While Reference [27] formalizes templates engines; our approach is focused on the definition of an Implementation Specific Meta-model and a generation process for the specific purpose of generating tag-based documents using a model-to-model transformation engine (ATL) [22] instead of a model-to-text transformation engines that are traditionally employed to generate this type of documents.

Unlike the proposal presented in Reference [28] where a concrete instantiation of a framework employing string, tree or triple graph grammars is required to generate source code; our approach only requires the definition of a model-to-model transformation to generate code because it addresses a more specific problem: the generation of tag-based documents only. As result, the generation process is simpler requiring only developers with skills EMF and ATL to generate code (i.e., any junior developer with basic skill in MDA can handle it).

We share the view presented in Reference [29] where authors propose the use of small Domain Specific Languages (DSLs) well-focused on meta-models, instead of monolithic languages like UML. The authors illustrate this approach to solve the bug-tracking by defining a meta-model to represent XML documents in Section 3.1. Although the meta-model presented in Reference [29] seems to be similar to the meta-model presented in this proposal, the purpose is completely different. While the goal of our approach is the definition of an Implementation Specific Model (ISM) to generate collections of XML documents for application development; the goal of the meta-model presented in Reference [29] is the definition of a generic document representation to pivot among different file formats. For instance, the meta-model presented in Reference [29] lacks on the capability to define file and folder structures to model the location of XML documents. This may not be a major issue if you are dealing with document formats; however, it is a minor issue in code generation because this characteristic is present in all model-to-text transformation languages. Finally, authors in Reference [29] do not provide any reference to the transformation that generates XML text documents, which is understandable because it is not the paper focus.

The work presented in Reference [30] describes a representation to convert model structures using model-to-model transformations. Although Section 3 of Reference [30] presents a similar process to tackle the problem of generating source code using model-to-model transformations instead of model-to-text transformation; the meta-model defined to carry out this task, which consists in a single meta-class (see Section 3.4 of Reference [30]), provides a high level of genericity at the expense of an overload of StringNode instance specification (e.g., the prefix and suffix definition is mandatory for all instances, including those that do not require any, such as plain text). The proposed solution is based on the definition of the meta-model presented in Section 3.1 of this article where XML document specific characteristics are abstracted to represent these documents employing less effort to generate text (e.g., no need to specify prefixes and suffixes for each tag or attribute). In addition, our approach enables developers to create XML document structures in terms of folders to organize the generated collection of XML documents.

Regarding Xtend language, this approach employs a Java dialect to describe transformations using an imperative programming paradigm instead of a declarative paradigm which is the same that is employed by most of model-to-model transformation languages. Thus, developers do not need to use different programming paradigms to generate XML documents.

Therefore, main article contributions can be summarized as follows:

- Enables developers to define the folder structure for the collection of XML documents to be generated according to a TagML model;

- Improves XML document generation inexpensiveness avoiding developers the need to learn model-to-text transformation languages (using a different programming paradigm) to generate XML documents;
- Reduces syntax errors derived from the definition of model-to-text transformations improving XML document generation accuracy;
- Improves transformation readability/understandability simplifying XML document representation;
- Improves transformation maintainability enabling developers to easily extend XML document model characteristics;
- Improves XML document generation predictiveness employing a unique model-to-text transformation.

This paper continues in Section 2 we highlight the main characteristics of collections of XML documents and the limitations of using traditional model-to-text transformation languages that adopt the archetype-based paradigm. Then, in Section 3 we describe the generation method for XML documents defined in different markup language formats. This section includes Section 3.1 to describe the TagML meta-model which captures the essential characteristics of documents defined in markup formats, and Section 3.2 that describes the definition of the model-to-text transformation that generates the textual representation of these documents in markup formats. A case study is presented in Section 4 to illustrate how the defined method is applied in a scenario where input models are defined by a subset of the entities defined in the Essential Meta-object Facility (EMOF) [31]. The discussion about pro and cons of using this approach to generate documents in markup format is exposed in Section 5. Finally, in Section 6 we present the conclusions and future works.

2. Motivation

This section describes the main characteristics of collections of XML documents and the limitations of using traditional model-to-text transformation languages using archetype-based paradigms. From the syntax perspective, these are the most relevant ones:

- The same tag keyword is used to define the beginning and the ending of XML document blocks. For instance, in XHTML, paragraphs are defined between the <P> and </P> tags where P represents the tag keyword.
- The same delimiters enclose keywords representing the beginning and the ending of XML document blocks. For instance, in XML, while the beginning of document blocks is defined by tag keywords enclosed by the <and> delimiters; the ending of document blocks is defined by tag keywords enclosed by the </and> delimiters.
- The shortened text representation for empty document blocks do not require end tags. For instance; the BR document block in XHTML is enclosed by the <and /> delimiters.
- The same delimiter is used to separate attributes in lists. For instance, the blank space is used to separate the rel = "stylesheet" and href = "/css/bootstrap.min.css" attributes in the XHTML rel = "stylesheet" href = "/css/bootstrap.min.css" attribute list.
- Attribute keys and values are separated by the same delimiters. For instance, in XHTML, the = character is used to separate the rel attribute key from the "stylesheet" attribute value.
- Attribute values are enclosed by the same delimiter. For instance, in XHTML, the " character is used to enclose the stylesheet attribute value.
- Collections of XML documents are usually structured in folders according to their contents

Most transformation languages that define model-to-text transformations are based on archetypes. Some examples of this type of languages are JET Templates [20], MOFScript [19] and ACCELEO [18].

These languages are very versatile because they enable developers to generate code for any programming language. For instance, the listing in Figure 1 depicts the except of code that defines a model-to-text transformation rule in ACCELEO to generate XHTML documents.

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/uml2/5.0.0/UML',
'http://www.eclipse.org/uml2/5.0.0/Types',
'http://www.eclipse.org/emf/2002/Ecore')]

[template public generateDocument(aClass : Class)]
[ file (aClass.name+'.html', false, 'UTF-8')]
<HTML lang="en">
<HEAD>[aClass.name/]</HEAD>
<LINK rel="stylesheet" href="./css/bootstrap.min.css"/>
<LINK rel="stylesheet" href="./css/bootstrap-theme.min.css"/>
<BODY>
<H1>[aClass.name/]</H1>
<P>Class [aClass.name/]</P>
<SCRIPT src="./js/jquery-3.1.1.min.js"></SCRIPT>
<SCRIPT src="./js/bootstrap.min.js"></SCRIPT>
</BODY>
</HTML>
[/ file]
[/ template]
```

Figure 1. The excerpt of the ACCELEO code that generates XHTML documents from UML class diagram models.

Figure 1 shows how the characteristics of XML documents are still present in the definition of model-to-text transformations using archetype-based languages that generate XML documents. The impact of these characteristics leads to information redundancy when defining model-to-text transformations.

The transformation rule depicted in Figure 1 requires developers to write tag keywords twice to define document blocks; for instance, to define the BODY block, developers have to explicitly write the <BODY> and </BODY> tags.

It also requires developers to write terminals for all opening, closing and empty tag keywords; for instance, the <, /> have to be explicitly introduced when defining tags such as <BODY>, </HTML> and <LINK/>.

Attribute list, attribute key-value and attribute value delimiters are required too. For instance, developers introduce spaces when defining the rel and href attributes of the <LINK rel = "stylesheet" href = "./css/bootstrap.min.css"/> document block. They also separate attribute keys from values using the = character (i.e., href = "./css/bootstrap.min.css"). And they enclose attribute values using quotes (i.e., "./css/bootstrap.min.css")

Another interesting issue to highlight about the definition of model-to-text transformations that generate XML documents using archetype-based languages is the fact that the generation of the textual representation of the XML document is inter-weaved with OCL expressions that calculates the information to be generated.

Under this scenario, the reuse of OCL expressions is poor because small changes in the syntax have a considerable impact in the definition of the transformation. For instance, suppose that the tag keyword delimiters changes from <and> to [and]. All transformations that have been defined using the <and> should be modified, or at least should be copied and modified if both versions of the transformation should be kept. This alternative present the same problems addressed by Model-driven architectures. For instance, the problem of divergent versions of the transformations when modifying model calculus [24].

As the result of the analysis of the peculiarities of the definition of model-to-text transformations using archetype-based languages, this proposal captures redundant information in XML document models following the Model-driven Architecture philosophy that leverages the abstraction and reuse principles to encourage the interoperability at design-time.

While the abstraction process captures XML document characteristics in the definition of a meta-model which is used to create XML document models; the reuse process is implemented by a model-to-text transformation that turns XML document models into text which generates redundant text.

Therefore, this approach enables developers to define model-to-model transformations instead of model-to-text transformations to generate XML text documents. It leads a clear separation between calculus and syntax where model calculus is captured by the model-to-model transformation, and syntax modifications are captured by the model-to-text transformation. For instance, to change <and> tag keyword delimiters by [and], developers modify, or copy and modify, the model-to-text transformation avoiding any change on the calculus. Conversely, to change model calculus, developers modify, or copy and modify, the model-to-model transformation, avoiding any change on the text generation.

Consequently, the proposed approach reduces the amount of information required to generate XML text documents which results on an increment of transformation definition reliability and a reduction of transformation maintenance costs.

3. The Method to Generate Documents

Figure 2 depicts the traditional process to generate XML documents using model-to-text transformations in archetype-based languages.

Figure 2. Traditional model-to-text transformation process.

The traditional process defines the $f_{m2t}(M_{PSM})$ model-to-text transformation function where M_{PSM} represents the Platform Specific Model (PSM) that should be turned into the text representation in XML format represented by *document*. Thus, f_{m2t} conforms to the $MM_{PSM} \rightarrow XML$ expression where MM_{PSM} represents the PSM meta-model and XML represents the XML format.

Conversely to the traditional approach, our proposal defines the process depicted in Figure 3.

Figure 3. The proposed model-to-text transformation process.

This proposal takes advantage of the XML document characteristics described in Section 2 to raise the level of abstraction while maximizing the reuse in the definition of model-to-text transformations that generate XML documents. Following the MDA philosophy, this proposal defines the Implementation Specific Model (ISM) abstraction layer described in Section 3.1 between the Platform Specific Model (PSM) and the textual representation of XML documents to decouple the text generation process from the model calculus.

This process is divided into two phases. The first phase of the transformation process is depicted in Figure 3. In this phase, transformation developers define the $f_{m2m}(M_{PSM})$ model-to-model

transformation function that turns the M_{PSM} PSM conforming the MM_{PSM} meta-model into the M_{ISM} ISM conforming the MM_{ISM} meta-model which captures the XML document characteristics described in Section 2.

The second phase of the transformation process is depicted in Figure 3. It defines the $f'_{m2t}(M_{ISM})$ model-to-text transformation function, which is defined in Section 3.2. This function turns the M_{ISM} ISM conforming the MM_{ISM} meta-model generated in the previous phase into the textual representation of the XML document the M_{ISM} ISM represents. It also creates the folder structure to organize generated XML documents.

Employing this approach, transformation developers focus on the development of the model-to-model transformation that defines model calculus leaving the syntax issues to the model-to-text transformation.

3.1. The Implementation Specific Meta-Model

The Implementation Specific Model (ISM) is defined in TagML. TagML is a Domain Specific Language (DSL) which was designed to capture XML-based documents characteristics to leverage the level of abstraction and reuse when defining model-to-text transformations that generate collections of XML documents. It is defined in terms of the meta-model depicted in Figure 4.

TagML was developed as an Eclipse platform feature using the Eclipse Modeling Framework (EMF) [32]. Consequently, this feature consists on a set of plugins that follow the Object Management Group (OMG) standards which ensure the interoperability of this tool with third party software. This way of developing tools presents some advantages with respect to those developed ad-hoc.

The capability of integrating this plugin with third part plugins (e.g., UML plugins, transformation tools, etc.), the capability of Eclipse plugins to be executed in different platforms (e.g., Windows, Linux, Mac, etc.), the functionality that is inherited from the Eclipse platform (e.g., workspace, project and file management, user configuration, customized text editors, plugin configuration management, task management, model repositories, etc.)

Both the TagML model editor and the M2T transformation have been developed as Eclipse plugins. These plugins were developed with the Eclipse Modeling Framework (EMF) technology. The use of EMF ensures that these tools follow the standards of the Object Management Group (OMG) ensuring their interoperability with third-party tools.

Developing tools such as Eclipse plugins has certain advantages with respect to the development of ad-hoc environments. The first one is to have a multi-platform development environment. In addition, we can mention the inherited functionality of the environment, which, for example, provides management of workspaces, projects, files, configuration for the user, custom text editors, integration with third-party tools, management of plugins configuration, management of tasks, model management, and so forth.

The goal of the TagML meta-model is to capture the characteristics of tag-based languages to raise the level of reuse of M2T transformations. Figure 4 shows the meta-model TagML.

As you can see in the Figure 4, this meta-model defines Archive as the root of a TagML model by defining a collection of FolderContent instances. The FolderContent, Folder and Document meta-classes define an instance of the Composite design pattern [33]. The role of FolderContent is that of Component, that of Folder is that of Composite and that of Document is that of Leaf. This design pattern is ideal for representing the structure of files and folders that contain the XML documents to be generated.

Each Document instance defines a Tag instance as the root label of the document. This instance is part of another instance of the design pattern Composite where the TagContent fulfills the role Composite, Text the role Leaf and Tag fulfills the role Composite. Again, this design pattern is best suited to represent the tree structure that all label-based documents define.

Text instances represent plain text that can be part of the content of a label. Tag instances represent a tag that defines a collection of attributes represented by Attribute instances.

Figure 4. The meta-model of TagML.

All meta-classes, except TagContent, define the attribute *name*, inherited from the Element meta-class abstract.

Figure 5 shows the example of a TagML instance model of a Web document in XHTML and its representation in a Web browser.

Figure 5. Example of the TagML model of a Web document.

3.2. Model-to-Text Transformation

The M2T transformation of a PSM is carried out in 2 phases:

1. The execution of the M2M transformation of the PSM to the TagML model;
2. The execution of the M2T transformation of the TagML model to the tag language.

In this section, we will focus on the M2T transformation that uses a model in TagML as an input model and, as an output, generates a document based on labels.

As mentioned above, TagML models are independent of the domain of the application. They only describe the content of the document to be generated.

The M2T transformation has been developed in ACCELEO. ACCELEO is an M2T transformation language based on archetypes.

The transformation module defines 7 templates (template) and 2 queries (query). The structure of document generation can be seen in the Figure 6.

Figure 6. XML document generation template in ACCELEO.

Figure 7 shows the code in ACCELEO of the transformation of an XML document described in a TagML model. This code template generates the file that will contain the XML document. The calculation of the file name is calculated with the query `getPath()`. In addition, it starts the code generation of the tags, starting with the root tag.

```

[query public getPath(aDocument : Document) : String =
if (self.parent.ocllsUndefined())
then '/' + self.archive.name else self.parent.getPath() endif /]
[query public getPath(aFolder : Folder) : String =
if (aFolder.parent.ocllsUndefined())
then '/' + self.archive.name else self.parent.getPath() endif + '/' + self.name /]
[template public generateDocument(aDocument : Document)]
[comment @main/]
[file (aDocument.getPath() + '/' + aDocument.name, false, 'UTF-8')]
[self.root.generateTag() /][file][template]

```

Figure 7. XML document generation template in ACCELEO.

The Figure 8 shows the code related to the generation of labels. This code is responsible for generating the code associated with a label, the attributes associated with it and the recursive call to the nested labels.

```

[template public generateTag(aTag : Tag) post (trim())
[if (aTag.content->isEmpty())<[aTag.name]/[aTag.generateAttributes()]/>
[else]<[aTag.name]/[aTag.generateAttributes()]/>
[for (tagContent : TagContent | aTag.content)]
[tagContent.generateContent() /][for][if]
[if (not aTag.content->isEmpty())<[/aTag.name]/>[/if][template]

```

Figure 8. Template for generating XML tags in ACCELEO.

To generate the code of the attributes, the code is shown in Figure 9.

The recursive call to generate the nested tags is carried out by calling the polymorphic function `generateContent` in `FolderContent`. This function which distributes its execution according to the type of element that is treated. If it is a label, then the function `generateTag` shown in Figure 8 is executed.

Otherwise, the function `generateText` that can be seen in Figure 10 is executed. This function generates the content associated with a plain text.

```
[template public generateAttributes(aTag : Tag)]
[for (attr : Attribute | aTag.attribute)][attr.generateAttribute() /][ /for ][ /template ]
[template public generateAttribute(attr : Attribute)]
[if (not attr.value.ocllsUndefined())][ self.name/]=' [ attr.value / ] '[ else ][ self.name/ ]
[/if]
[/template]
```

Figure 9. Template for generating attributes of an XML tag in ACCELEO.

```
[template public generateText(aText : Text) post (trim())][ self.content / ][ /template ]
```

Figure 10. Plain text generation template in ACCELEO.

3.3. Implementation Issues

Both, the TagML model editor and the M2T transformation, have been developed as Eclipse plugins. These plugins were developed with the Eclipse Modeling Framework (EMF) technology. The use of EMF ensures that these tools follow the standards of the Object Management Group (OMG) ensuring their interoperability with third-party tools.

Developing tools such as Eclipse plugins has certain advantages with respect to the development of ad-hoc environments. The first one is to have a multi-platform development environment. In addition, we can mention the inherited functionality of the environment, which, for example, provides management of workspaces, projects, files, configuration for the user, custom text editors, integration with third-party tools, management of plugins configuration, management of tasks, model management, and so forth.

3.4. Extensions

The TagML meta-model enables developers to represent XML documents that are part of a document structure where XML documents are transformed into files organized in folders. This meta-model defines two types of document contents: tags and free text. However, XML documents allow other types of content other than just tags or free text (e.g., comments, cdata, xml version, etc.) which are not explicitly defined in the meta-model. This fact leads to the use of free text type to represent these characteristics; for instance, you can represent a comment using an instance of `Text` which content is `<!--Comment-->`. Although it is a valid approach; it is not a good one because the advantages of using TagML are not applied to these elements.

A good approach to solve this issue begins with the definition of new sub-meta-class of the `TagContent` meta-class (e.g., `Comment`) defining the `content:EString` attribute to represent the comment contents (without the stating and ending marks). Then, it continues with the overload of the `generateContent` template method following the pattern defined in Figures 10 and 11 as seen on Figure 12. Finally, regenerate EMF plugin and compile the model-to-text transformation to support comments. Note that the same approach can be used to extend the language to any structure.

```
[template public generateContent(aTagContent : TagContent)
? (aTagContent.ocllsTypeOf(Tag)) post (trim())]
[aTagContent.ocllsTypeOf(Tag).generateTag() / ][ /template ]
[template public generateContent(aTagContent : TagContent)
? (aTagContent.ocllsTypeOf(Text)) post (trim())]
[aTagContent.ocllsTypeOf(Text).generateText() / ][ /template ]
```

Figure 11. Template for generating the content of an XML tag in ACCELEO.

```

[template public generateContent(aTagContent : TagContent)
? (aTagContent.ocllsTypeOf(Comment)) post (trim())]]
[aTagContent.ocllAsType(Comment).generateComment() /][template]

[template public generateComment(aComment : Comment) post (trim())]]
<!-- [self.content/] -->[/template]

```

Figure 12. Comment generation template extension to TagML in ACCELEO.

4. Case Study

Once defined the meta-model that allows to model collections of XML documents, and the transformation that allows the generation of the code associated with these models, we proceed to show a case study to demonstrate the versatility of the proposed generation method.

The case study described in this section, is a generator of Web documents in XHTML from a UML class model [34,35].

Figure 13 presents an example of an input model that defines 3 classes. This model has been generated with the Papyrus tool [36]. Figure 14 shows the result of applying the transformation method proposed in the Section 3.

Figure 13. UML class diagram.

Figure 14. XHTML generated documents.

As seen in the Section 3, the process defines two phases. While the first phase requires the definition of an M2M transformation of the PSM to a TagML model, the second one does not require any intervention from the developer being fully automatic and independent of the PSM. Therefore, we will focus on the definition of the first transformation.

The first step in defining the first transformation is to identify the meta-input and output models. In this case they are UML and TagML.

To define this transformation we used ATL [22] which is a declarative language to define model-to-model transformations. However, you can use any other, for example, QVT [21]. The reason for this is that the model-to-text transformation is carried out from the TagML model.

The code shown in Figure 15 converts the instance of Model into an Archive instance and the Package instance into Folder instances.

```
rule Model2Archive {
  from model: UML!Model to folder: TagML!Folder
  (name <- model.name.firstToLower(),
   archive <- archive),
  archive: TagML!Archive (name <- model.name.firstToLower())}

rule Package2Folder {
  from model: UML!Package (model.ocllsTypeOf(UML!Package)) to folder: TagML!Folder
  (name <- model.name.firstToLower(),
   parent <- model.namespaces)}
```

Figure 15. Transformation rules from Model to Archive and from Package to Folder in ATL.

Figure 16 shows the most relevant lines of code of how classes are converted to documents.

```
rule Class2Document {
  from model: UML!Class to form: TagML!Tag
  (name <- 'FORM', parent <- divContainerCol),
  document: TagML!Document (
  name <- model.name.firstToLower() + '.html',
  parent <- model.namespaces),
  html: TagML!Tag (name <- 'HTML', document <- document),
  head: TagML!Tag (name <- 'HEAD', parent <- html),
  body: TagML!Tag (name <- 'BODY', parent <- html),
  p: TagML!Tag (name <- 'H1', parent <- form),
  pText: TagML!Text (
  content <- 'Formulario '.concat(model.name.firstToUpper()),
  parent <- p)
}
```

Figure 16. Rule of transformation of Class a Document in ATL.

The code in Figure 17 shows the rule that transforms the properties of a class into the fields of the form associated to the class.

```
abstract rule Attribute2Field {
  from property: UML!Property (property.association.ocllsUndefined())
  using { inputIdString: String = property.name.concat('_ID'); }
  to label: TagML!Tag (name <- 'LABEL', parent <- formGroup),
  labelText: TagML!Text ( content <- property.name.firstToUpper().concat(': '),
  parent <- label),
  labelFor: TagML!Attribute (name <- 'for', value <- inputIdString, tag <- label),
  input: TagML!Tag (name <- 'INPUT', parent <- formGroup),
}
```

Figure 17. Property to Tag transformation rule in ATL.

The model resulting from executing the transformations can be seen in Figure 18. The result of the M2M transformation can be seen in the left half of Figure 18. In it, you can see that the root node is an instance of Archive. This node contains the model folder, which is an instance of Folder. Inside the folder model you can see three documents (Document instances): invoice.html, linea.html and client.html. The content of the document factura.html defines a set of labels that make up its structure. In this case, we can see the labels that present the branch of the tree that contain the form that corresponds to the

class Invoice. An extract of the result of the M2T transformation is shown in the left half of Figure 18. In this extract you can see the correspondence of the Tag instances and the generated XML tags.

For instance, an example of the benefit of employing this approach from the maintainability perspective is the simplicity of changing tag delimiters (i.e., “<”, “>”, “/>” by “[”, “]” and “/”) by duplicating the proposed transformation and performing a small modification of 4 characters in lines 2, 3 and 6 of the code presented in Figure 8. This fact reduces maintainability costs because all models represented with the proposed meta-model can be reused to generate the same documents using a different format without changing XML document models at all.

Figure 18. Results of the model-to-model (M2M) and M2T transformations.

5. Evaluation of the Proposed Method

This section discusses the evaluation of the proposed method to tackle the generation of XML documents using M2M transformations. This evaluation compares the use of the traditional mechanism (archetype based M2T transformation languages), against declarative M2M transformation languages for the definition of model transformations that generate XML documents.

The evaluation is performed in two steps: first a quantitative evaluation is described, then a qualitative evaluation is presented, both to compare the two techniques.

5.1. Quantitative Evaluation

The goal of the quantitative evaluation is comparing the amount of “unhandled” code required to generate XML documents using the TagML approach to the amount of “unhandled” code required to generate XML documents using a traditional approach (i.e., using a model-to-text transformation); where the term “unhandled” code refers to the code required to generate XML documents that are

not part of the transformation language (i.e., it is not automatically generated by the transformation process). As the “unhandled” code is not validated in any way, the more “unhandled” code is required to generate a document, the more error prone it is.

The first step to perform this comparison is the definition of a common set characteristics that are required by both approaches in order to factorize then and remove them from the comparison. The following set of equations define the set of common characteristics of XML documents that are present in both approaches:

$$I = \{\text{valid XML Strings to represent identifiers}\} \quad (1)$$

$$S = \{\text{valid XML Strings to represent text}\} \quad (2)$$

$$A = \{(n, v); n \in I \wedge v \in S\} \quad (3)$$

$$T = \{(t, A', T'); t \in I \wedge \forall a' \in A', a' \in A \wedge \forall t' \in T', t' \in T\} \quad (4)$$

$$D = \{(d, T''); d \in I \wedge \forall t'' \in T'', t'' \in T\}, \quad (5)$$

where

- I represents the set of valid XML Strings to define XML identifiers (e.g., do not contain any spaces);
- S represents the set of valid XML Strings to define XML text (e.g., representation of accentuated characters in ASCII);
- A represents the set of valid XML attributes where n and v represent attribute names and values respectively;
- T represents the set of valid XML tags where A' represents the set of attributes of the tag t and T' represents the set of tags that are nested in the tag named t ;
- D represents the set of valid XML documents where d represents an XML document name and T'' represents the set of elements defined at the top level of the XML document named d .

Once common characteristics were defined, the amount of “unhandled” generated code is analyzed according to 3 cases based on the definition of the set of transformation rules used to generate XML document attributes, empty tags, and composed tags.

5.1.1. Case 1: Attribute Generation

The Equation (3) defines the set of valid attributes in XML. The code required to generate attributes in an archetype-based M2T transformation language (i.e., ACCELEO) is $! [n/] = "[v/]" !$. And the amount of “unhandled” code to generate these tags is 4 (i.e., $! = "" !$).

If $f_A(d)$ represents the set of valid XML attributes for the XML document d , $C_A(d) = 4 \times \#f_A(d)$ is the amount of “unhandled” code.

Alternatively, the code required to generate attributes in a declarative M2M transformation language (i.e., ATL) is $! \text{attr} : \text{TagML!Attribute} (\text{name} <- n, \text{value} <- v) !$. In this case, all the code is part of the transformation language; therefore, the amount of “unhandled” code is 0.

5.1.2. Case 2: Empty Tag Generation

According to the Equation (4), $T_{C2} = \{(t, \emptyset, \emptyset); t \in I\}$ defines the set of empty tags without attributes.

The code required to generate empty tags without attributes in an archetype-based M2T transformation language (i.e., ACCELEO) is $! <[t/] /> !$. And the amount of “unhandled” code to generate these tags is 3 (i.e., $! < /> !$).

If $f_{ET}(d)$ represents the set of empty tags without attributes of an XML document d , $C_{ET}(d) = 3 \times \#f_{ET}(d)$ is the amount of “unhandled” code.

On the other hand, the code required to generate empty tags without attributes in a declarative M2M transformation language (i.e., ATL) is `| tag : TagML!Tag (name <- t) |`. Again, all the code is part of the transformation language; therefore, the amount of “unhandled” code is 0.

5.1.3. Case 3: Composed Tag Generation

According to the Equation (4), $T_{C3} = \{(t, \emptyset, T'); t \in I \wedge \forall t' \in T' (t' \in T) \wedge T' \neq \emptyset\}$ defines the set of composed tags without attributes. The code required to generate empty tags without attributes in an archetype-based M2T transformation language (i.e., ACCELEO) is `!<[t]>[for t' in T'] [/for]</[t/]>!`. And the amount of “unhandled” code to generate these tags is 5 (i.e., `!<></>!`).

If $f_{CT}(d)$ represents the set of composed tags without attributes of an XML document d , $C_{CT}(d) = 5 \times \#f_{CT}(d)$ is the amount of “unhandled” code.

However, if the tag name (i.e., t) is not generated in a local variable, and it is a constant value, t should be written twice. Therefore, $C'_{CT}(d) = 5 \times (\#f_{CT}(d) + l(t))$ where $t \in f_{CT}(d)$ and $l(x)$ represents the length of x .

On the other hand, the code required to generate empty tags without attributes in a declarative M2M transformation language (i.e., ATL) is `| tag : TagML!Tag (name <- t) |`. Again, all the code is part of the transformation language; therefore, the amount of “unhandled” code is 0.

Instead, the code required to generate composed tags without attributes in a declarative M2M transformation language (i.e., ATL) is `| tag : TagML!Tag (name <- t, contents <- T'.collect(t' | )) |`. Again, all the code is part of the transformation language; therefore, the amount of “unhandled” code is 0.

5.1.4. Analysis

The Figure 19 shows the evolution of the “unhandled” code using both approaches. On the one hand, Figure 19a shows the lineal evolution of the amount of “unhandled” code using a M2T transformation language in terms of the number of attributes and empty tags. On the other hand, Figure 19b shows the evolution of the amount of “unhandled” code using a M2T transformation language in terms of the amount of composed tags (x axis) and the length of tag names (y axis).

Figure 19. The evolution of the amount of “unhandled” code

5.2. Qualitative Evaluation

This validation is based on the five general dimensions of Model Driven Engineering (MDE) [37] that a good model should cover. And it is based on the application of TagML in 3 different scenarios besides the presented case of study. These scenarios include—a set of 7 student projects for the subject “Advanced software development methods” (4th year of Computer Science degree) and a final degree project in 2017 [23], and a PhD thesis [38], which is based on the meta-models presented in Reference [39].

In Reference [37], the authors explain that a good model covers the following five characteristics:

1. **Abstraction:** A model should reduce the details of the system it represents.
2. **Understandability:** A model should be easier to understand than the system it models.
3. **Accuracy:** While abstraction may summarize or hide important details, the meaning of these details should not be altered by a model.
4. **Predictiveness:** A model should be executable in order to help developers predict specific system behavior.
5. **Inexpensiveness:** A model must be inexpensive to produce.

5.2.1. Abstraction

The definition of TagML models hides XML document syntax from developers while keeping model XML document structure. To carry out this task, this proposal defines an Implementation Specific Models (ISMs) to generate XML documents.

The definition of ISMs facilitates the definition of vertical transformations (to perform model abstraction and refinement), horizontal transformations (to perform model migrations and representations), and mixed transformations. For instance, an ISM enables developers to define abstraction transformations using ATL; instead of using a traditional parser that reads text and creates models manually. Moreover, the use of a declarative language, instead of an archetype-based language, enables developers to execute inverse transformations.

The use of ISMs also maximizes the interoperability at design time delaying the generation process to later stages of the development enabling developers to introduce changes in the TagML model that do not affect the rest of the models. For instance, developers can modify or add tags to XML documents in the TagML model avoiding changes in the transformation function definition improving its maintainability.

5.2.2. Understandability

This proposal decouples XML document syntax from its structure defining TagML models.

The use of TagML models improves transformation model understandability because it enables developers to focus on the XML document structure; instead of the XML syntax (terminals, closing tags, etc.) that is a key issue when using M2T archetype-based transformation approaches.

5.2.3. Accuracy

As we mentioned in Section 5.2.1, the definition of TagML models hides XML document syntax from developers while keeping model XML document structure. The proofs of accuracy are explained in Section 3.2 and depicted in Figures 7–11 where we present most relevant excerpts of code that define M2T transformation functions that transform TagML meta-model entities depicted in Figure 4 into XML document source code.

5.2.4. Predictiveness

The definition of TagML models improves the prediction of specific system behavior introducing XML document validation at meta-model level; for instance, to check that XML tag names are valid (e.g., do not contain any spaces). Thus, the XML document validation is defined once for all TagML models, improving model reliability, reuse and reducing maintenance costs; for instance, new invariants can be easily introduced without modifying the transformation function definition. It improves model predictiveness because model validation ensures that generated XML documents are syntactically valid.

Conversely, the use of a traditional approaches using M2T archetype-based transformation languages implies the definition of a validation function (e.g., query in ACCELEO) that must be called explicitly in all transformations whenever XML elements are generated. This practice interweaves the

validation code with the text generation, which MDA practitioners should avoid for the sake of the separation of concerns (i.e., XML document generation and validation).

5.2.5. Inexpensiveness

The definition of the TagML meta-model also enables developers to use the same transformation language to manipulate models and generate source code; instead of the two different languages that are employed in traditional approaches (archetype-based and declarative). It makes the generation process less expensive compared to traditional approaches where developers require skills in two transformation languages following different programming paradigms.

5.3. Threats to Validity

There are always threats to the validity of a study (e.g., in References [40,41]). This section analyzes these threats and describes the strategies used to mitigate their effects, as well as the limitations of this study. To assess the validity of this study, the authors used the validity framework presented by Reference [40] characterized by:

1. Construct validity
2. External validity
3. Internal validity
4. Conclusion validity

5.3.1. Construct Validity

The construct validity is related to the correctness of the measures to analyze the concept to be studied [40,41]. The test was applied to a concrete case of study that generates XHTML. This code does not include any sections of the document containing comments, version declarations, or character data; though these aspects were addressed in Section 3.4 where different types of contents can be easily managed following a pattern.

5.3.2. External Validity

The external validity is related to the degree of generalization of the results [40] where it is extremely important to describe the context of the research [42,43]. This main issue regarding this threat is the level of experience of the validation experiment participants. Although they all had experience in the development of model-to-text and model-to-model transformations they were mostly advanced degree students; they were not senior developers. It is a good scenario to check the approach learnability, which is one of the goal pursued by this approach; however, to generalize this study, this effect can be mitigated performing the validation with senior developers to compare results.

5.3.3. Internal Validity

The internal validity is related to causal relationships not measured during the experiment or the researchers did not control. It is related to the study performed in Section 5.1 where we employ descriptive statistics to analyze this method. The main thread regarding this issue is the limitation of the elements covered by the case of study (i.e., attributes, empty tags, and composed tags) which do not cover other elements such as comments, character data, version declarations, and so forth.

5.3.4. Conclusion Validity

The conclusion validity is related to the researcher interpretation of results. Although this risk can not be eliminated, it was reduced by taking the following measures:

- 2 researchers and 7 advanced degree students and 1 PhD student participated in the qualitative validation,

- A complete “audit” of the results was performed by 2 researchers and 1 advanced degree student and 1 PhD student,
- 10 participants in the study validated this approach.

6. Conclusions and Future Work

The first conclusion we can observe is that the proposed M2T transformation definition method facilitates a more readable, compact definition, with lower maintenance costs and less prone to errors than traditional methods.

In addition, it provides a complete development environment based on Eclipse plugins that allows easy integration with third-party tools. Also, it provides a set of services inherent to the platform that are available in the tool. Regarding the standards, when developing a plugin with EMF technology, all the devices comply with the OMG standards. Last but not least, this environment provides a system for updating and managing versions of the plugins that allow easy maintenance and extension of the tool.

As future work, we are working on extending the environment to customize and optimize M2T transformations that have a TagML model as input. We are also working on a new version of TagML to support meta-labels and markup languages that allow customizing transformations.

Author Contributions: Formal analysis, R.T.; Research, J.A.G., and G.S.; Methodology, R.T.; Software, G.S.; Supervision, R.T. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been partially supported by the national project granted by the Ministry of Science, Innovation and Universities (Spain) with reference RTI2018-099942-B-I00 and by the project TecnoCRA (ref: SBPLY/17/180501/000495) granted by the regional government (JCCM) and the European Regional Development Funds (FEDER).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Object Management Group. XML Metadata Interchange (XMI) Specification Version 2.5.1. 2015. Available online: <https://www.omg.org/spec/XMI/2.5.1> (accessed on 1 June 2020).
2. World Wide Web Consortium. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). 2002. Available online: <https://www.w3.org/TR/xhtml1/> (accessed on 1 June 2020).
3. World Wide Web Consortium. HTML 5.2. 2017. Available online: <https://www.w3.org/TR/html52/> (accessed on 1 June 2020).
4. Chung, K.M. JavaServer Pages 2.3 Specification. Technical report, Oracle. 2013. Available online: <https://jcp.org/aboutJava/communityprocess/final/jsr245/index.html> (accessed on 1 June 2020).
5. Burns, E.; Riem, M. JavaServer Faces Specification. Technical report, Oracle. 2017. Available online: <https://jcp.org/aboutJava/communityprocess/pfd/jsr372/index.html> (accessed on 1 June 2020).
6. Java Community Process. JSR-52: A Standard Tag Library for JavaServer Pages. 2006. Available online: <https://jcp.org/aboutJava/communityprocess/maintenance/jsr052/index3.html> (accessed on 1 June 2020).
7. Microsoft. ASP .NET Web Site. 2017. Available online: <https://www.asp.net/> (accessed on 1 June 2020).
8. Microsoft. XAML Documentation. 2012. Available online: <https://msdn.microsoft.com/en-us/library/gg134030.aspx> (accessed on 1 June 2020).
9. Google. Polymer Project. 2017. Available online: <https://www.polymer-project.org/1.0/> (accessed on 1 June 2020).
10. Google. AngularJS. 2017. Available online: <https://angularjs.org/> (accessed on 1 June 2020).
11. ISO. X3D V3.3 Abstract Specification; ISO ISO/IEC IS 19775-1:2013; International Organization for Standardization: Geneva, Switzerland, 2013.
12. Android. Android Web Site. 2017. Available online: <https://www.android.com/> (accessed on 1 June 2020).
13. World Wide Web Consortium. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). 2007. Available online: <https://www.w3.org/TR/soap12-part1/> (accessed on 1 June 2020).
14. Object Management Group. OMG Web Site. 2017. Available online: <http://www.omg.org/> (accessed on 1 June 2020).

15. Object Management Group. MDA—The Architecture Of Choice For A Changing World. 2017. Available online: <http://www.omg.org/mda/> (accessed on 1 June 2020).
16. World Wide Web Consortium. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. 2012. Available online: <https://www.w3.org/TR/xmlschema11-1/> (accessed on 1 June 2020).
17. World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0. 1999. Available online: <https://www.w3.org/TR/xslt> (accessed on 1 June 2020).
18. Eclipse Foundation. Obeo. Acceleo. 2017. Available online: <http://www.eclipse.org/acceleo/> (accessed on 1 June 2020).
19. Eclipse Foundation. MOFScript Home Page. 2011. Available online: <https://eclipse.org/gmt/mofscript/> (accessed on 1 June 2020).
20. Eclipse Foundation. Java Emitter Templates (JET2). 2011. Available online: <https://projects.eclipse.org/projects/modeling.m2t.jet> (accessed on 1 June 2020).
21. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). 2016. Available online: <http://www.omg.org/spec/QVT/1.3> (accessed on 1 June 2020).
22. Jouault, F.; Allilaire, F.; Bézivin, J.; Kurtev, I. ATL: A model transformation tool. *Sci. Comput. Program.* **2008**, *72*, 31–39. [CrossRef]
23. Fardoun, H.M.; Tesoriero, R.; Sebastián, G.; Safa, N. A Simplified MbUID Process to Generate Web Form-based UIs. In Proceedings of the 13th International Conference on Software Technologies, ICSOFT 2018, Porto, Portugal, 26–28 July 2018; Maciaszek, L.A., van Sinderen, M., Eds.; SciTePress: Setubal, Portugal, 2018; pp. 835–842. [CrossRef]
24. Mellor, S. *MDA Distilled, Principles of Model Driven Architecture*; Addison-Wesley Professional: Boston, MA, USA, 2004.
25. Kolovos, D.S.; Rose, L.M.; Williams, J.; Matragkas, N.; Paige, R.F. A Lightweight Approach for Managing XML Documents with MDE Languages. In *Proceedings of the 8th European Conference on Modelling Foundations and Applications, ECMFA'12*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 118–132. [CrossRef]
26. Kolovos, D.S.; Paige, R.F.; Polack, F.A.C. The Epsilon Object Language (EOL). In *Model Driven Architecture—Foundations and Applications*; Rensink, A., Warmer, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2006; pp. 128–142.
27. Parr, T.J. Enforcing Strict Model-view Separation in Template Engines. In *Proceedings of the 13th International Conference on World Wide Web, WWW '04*; ACM: New York, NY, USA, 2004; pp. 224–233. [CrossRef]
28. Anjorin, A.; Saller, K.; Rose, S.; Schürr, A. A Framework for Bidirectional Model-to-Platform Transformations. In *Software Language Engineering*; Czarnecki, K., Hedin, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 124–143.
29. Bézivin, J.; Bruneliere, H.; Jouault, F.; Kurtev, I. Model engineering support for tool interoperability. In Proceedings of the 4th Workshop in Software Model Engineering (WiSME 2005), Montego Bay, Jamaica, 4 July 2005.
30. Willink, E.D. A text model—Use your favourite M2M for M2T. In Proceedings of the 18th International Workshop in OCL and Textual Modeling (OCL 2018), Copenhagen, Denmark, 14 October 2018.
31. Object Management Group. Meta Object Facility (MOF) 2.5.1. 2016. Available online: <http://www.omg.org/spec/MOF/2.5.1/> (accessed on 1 June 2020).
32. Eclipse Foundation. EMF Eclipse Modeling Framework (EMF). 2017. Available online: <https://www.eclipse.org/modeling/emf/> (accessed on 1 June 2020).
33. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns Elements of Reusable Object-Oriented Software*; Addison-Wesley: Reading, MA, USA, 2000.
34. Object Management Group. Unified Modeling Language (UML). 2015. Available online: <http://www.omg.org/spec/UML/2.5> (accessed on 1 June 2020).
35. Rumbaugh, J.; Jacobson, I.; Booch, G. *Unified Modeling Language Reference Manual*, 2nd ed.; Pearson Higher Education: New York, NY, USA, 2004.
36. Eclipse Foundation. Papyrus Modeling Environment. 2017. Available online: <https://eclipse.org/papyrus/> (accessed on 1 June 2020).
37. Selic, B. The Pragmatics of Model-Driven Development. *IEEE Softw.* **2003**, *20*, 19–25. [CrossRef]
38. Sebastián, G.; Tesoriero, R.; Gallud, J.A. Automatic Code Generation for Language-Learning Applications. *IEEE Latin Am. Trans.* **2020**, *18*, 1433–1440. [CrossRef]

39. Sebastián Rivera, G.; Tesoriero, R.; Gallud, J.A. Model-based approach to develop learning exercises in language-learning applications. *IET Softw.* **2018**, *12*, 206–214. [[CrossRef](#)]
40. Wohlin, C.; Runeson, P.; Hst, M.; Ohlsson, M.C.; Regnell, B.; Wessln, A. *Experimentation in Software Engineering*; Springer Publishing Company, Incorporated: New York, NY, USA, 2012.
41. Runeson, P.; Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* **2008**, *14*, 131. [[CrossRef](#)]
42. Petersen, K.; Wohlin, C. A comparison of issues and advantages in agile and incremental development between state of the art and an industrial case. *J. Syst. Softw.* **2009**, *82*, 1479–1490. [[CrossRef](#)]
43. Kitchenham, B.A.; Budgen, D.; Brereton, O.P. Using mapping studies as the basis for further research—A participant-observer case study. *Inform. Softw. Technol.* **2011**, *53*, 638–651. [[CrossRef](#)]

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).